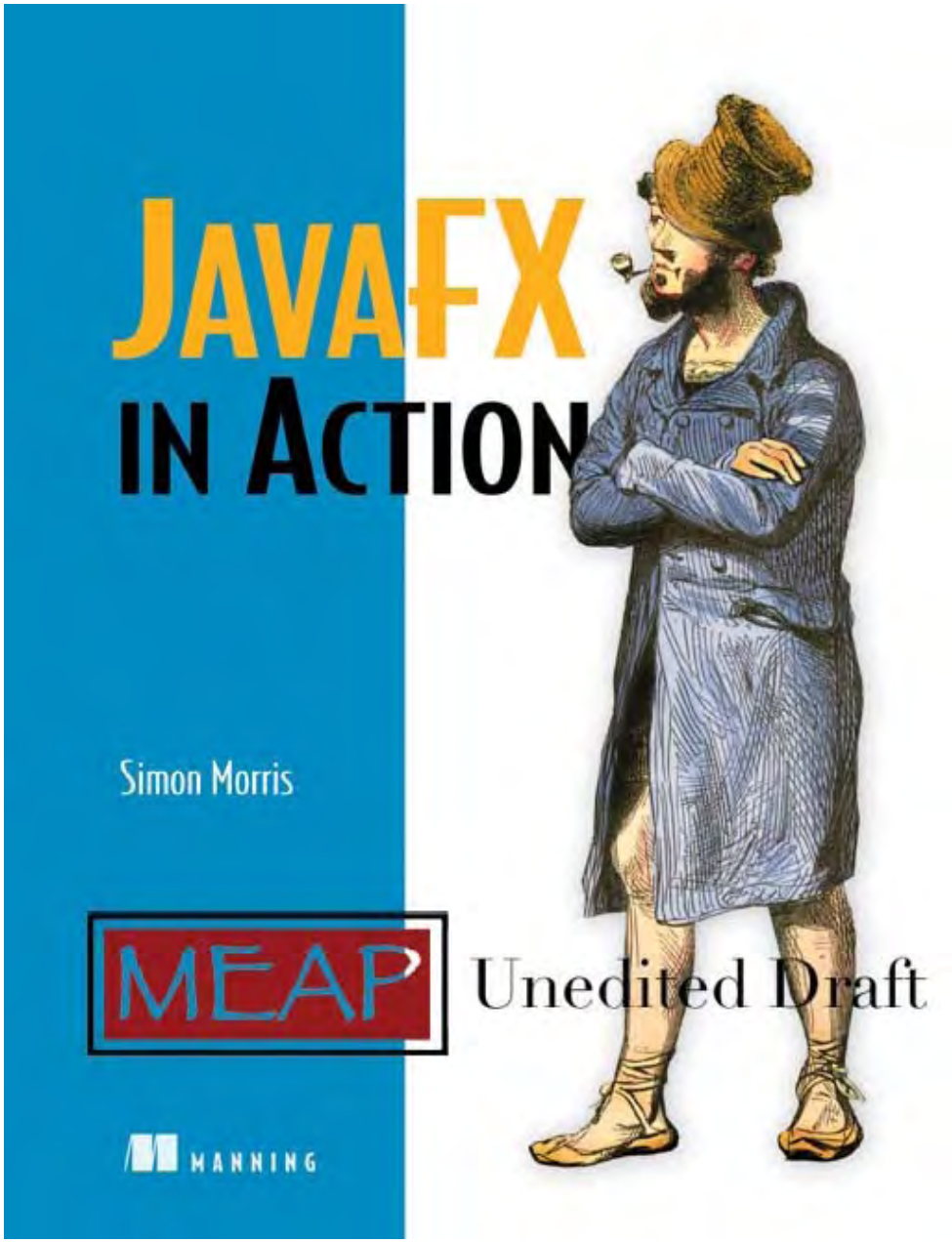


©Manning Publications Co. Please post comments or corrections to the Author Online forum:

MEAP Edition
Manning Early Access Program

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=498

1

Welcome to the future: introducing JavaFX

"*When the only tool you have is a hammer, every problem looks like a nail*", the popular saying goes.

Language advocacy is a popular pastime with many programmers, but what many fail to realize is that programming languages are like tools: each is good at some things and next to useless at others. Java, inspired as it was by prior art like C and Smalltalk, sports a solid general purpose syntax which gets the job done with the minimum of fuss in the majority of cases. Unfortunately there will always be those areas which, by their very nature, demand something a little more specialized. Graphics programming has typically been one such area.

Graphics programming used to be fun! Early personal computer software sported predominantly character based user interfaces. Bitmap displays were too expensive, although some computers offered the luxury of hardware sprites. For the programmer, the simple act of poking values into RAM gave instant visual gratification.

These days things are a lot more complicated; we have layers of abstraction separating us from the hardware. Sure, they give us the wonders of scrollbars, rich text editors and tabbed panes, but they also constrain us. The World Wide Web raised the bar; users now expect glossier visuals, yet the tools that we programmers use to create desktop software are in essence little evolved from the days of the first Macintosh or Amiga.

But it's not just the look of software which has been changed by the web. Increasingly data is moving away from the hard disk and onto the internet. Our tools are also starting to move that way, yet the fledgling attempts to build on-line applications using HTML and Ajax have resulted in nothing more than pale imitations of their desktop cousins. At the same time consumer devices like phones and TV set top boxes are getting increasingly

sophisticated in terms of their UI, and faster wireless networks are reaching out to these devices, allowing applications to run in places previously unheard of.

If only there were a purpose built tool for writing the next generation of internet software, one which could serve up the same rich functionality of a desktop application, yet with drop-dead gorgeous visuals and rich media content within easy reach, delivered to whatever device (PC or smart phone) we wanted to work from today.

Sounds too good to be true? Let me introduce you to JavaFX!

1.1 Introducing JavaFX

JavaFX is the name of a family of technologies for developing visually rich applications across a variety of devices. Version 1.0 was launched in December 2008, focusing on the desktop and web applets. Version 1.1 arrived a couple of months later, adding phone support to the mix. Later editions promise to expand the platform's repertoire to TV devices, like personal video recorders or perhaps Blu-ray disc software, and enhance its desktop support beyond just animation and multimedia to include next-gen UI controls.

The JavaFX APIs consists of a radically different way of handling graphics, known as *retained mode*, shifting focus away from the pixel pushing *immediate mode* (ala Swing), towards a more structured approach which makes animation cleaner and easier. At JavaFX's center is a major new programming language, *JavaFX Script*, built from the ground up for modeling and animating multimedia applications. JavaFX Script is compiled and Object Oriented, with a syntax independent of Java, but capable of working with Java created class files. Together JavaFX Script (the language) and JavaFX (the APIs and tools) create a modern, powerful and convenient way to create software.

1.1.1 Why do we need JavaFX Script? The power of a 'DSL'

A very good question; why **do** we need yet another language? The World is full of programming languages, wouldn't one of the existing languages do? Perhaps JavaScript, or Python, or Scala? Indeed, what's wrong with Java? Certainly JavaFX Script makes writing slick graphical applications easier, but is there more to it than that?

What makes graphics programming such an ill fit for modern programming languages? There are many problems, ask a dozen experts and you'll get thirteen answers, but let me (your humble author) risk suggesting a couple of prime suspects.

Figure 1.1 A complex GUI typical of modern desktop applications. Two windows host scrolling control palettes, while another holds an editable image and rulers.

Firstly, graphics generally require quite large nested data structures: trees of elements, each element providing a baffling array of configurable options and behaviors. Procedural languages like to work in clearly delineated steps, but this linear pattern conflicts with the tree pattern inherent in most GUIs.

Secondly, graphics code tends to rely heavily on concurrency–processes running in parallel. Modern UI fashions have amplified this requirement, with several transition effects often running within a single interface simultaneously. The boilerplate code demanded by many languages to create and manage these animations is verbose and cumbersome.

Perhaps you can think of other problems, but the above two (at least in my experience) seem to create more than enough trouble between them. It's deep, fundamental, problems like these that a *domain specific language* can best address.

A DSL is a programming language designed from the ground up to meet a particular set of challenges, and solve a specific type of problem. The language at the heart of JavaFX, JavaFX Script, is an innovative DSL for creating visually rich user interfaces. It boasts a declarative syntax, the code structure mirrors the structure of the interface; associated UI components are kept in one place, not strewn across multiple locations. Simple language constructs soothe the pain of updating and animating the interface, reducing code complexity while increasing productivity. The language syntax is also heavily expression based, allowing tight integration between *object models* and the code which controls them.

In layman's terms, JavaFX Script is a tool custom made for user interface programming.

But JavaFX isn't just about slick visuals; it's also an important weapon in the arms race for the emerging Rich Internet Application market. But what is an RIA?

1.1.2 Back to the future: the rise of the cloud

Douglas Adams once wrote "*I suppose the best way to find out where you come from is to find out where you're going, and then work backwards.*"

Sometimes we become so engrossed in the here and now, we forget to stop and consider how we arrived at where we are. We know where we want to go, but can our past better help us get there?

In the pre-internet age software was installed straight onto the hard drive. If suddenly overcome by an urge to share with friends your latest poetic masterpiece, you needed to hand both the document file and the software to open it. Chances are neither would be available. Your friends might be grateful, but clearly this was a problem needing a solution.

The World Wide Web was a small step towards that solution. Initial *applications* were nothing more than query/response database lookups, but web mail changed all that. Web mail marked a fundamental shift in the relationship between site and visitor. Previously the site held content which the visitor browsed or queried; but web mail sites supplied no content themselves, relying instead on content from (or for) the user. The role of the site had moved from information source, to storage depot, and the role of the visitor from passive consumer to active producer.

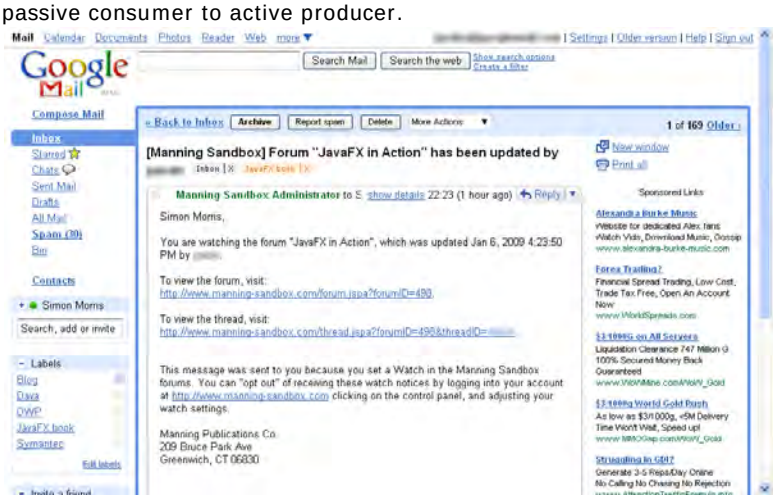


Figure 1.2 Google's Gmail is an example of a web site application which attempts to mimic the look and function of a desktop application.

A new generation of web sites attempted to ape the look and feel of traditional desktop software, earning the moniker "*Rich Internet Application*" after Macromedia (subsequently purchased by Adobe) coined the term in a 2002 white paper noting the transition of applications from the desktop onto the web. By late 2007 the term *cloud computing* was in

common use to describe the anticipated move from the hard disk to the network for storing personal data, like word processor documents, music files or photos.

Despite the enthusiasm, progress was slow and frustrating. Ajax helped paper over some of the cracks, but at its heart the web was designed to show page based content, not run software. Web content is 'poured' into the window, left to right down the page, echoing the technology's publishing origins, while input is predominantly restricted to basic form components. Mimicking the layout and functionality of a desktop application inside a document-centric environment was not easy, as numerous web developers soon discovered.

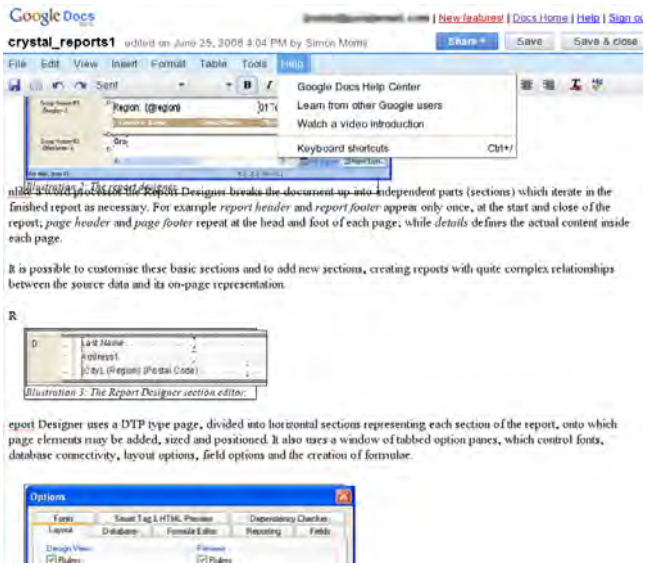


Figure 1.3 Google Docs runs inside a browser and has a much simpler GUI than Microsoft Office or OpenOffice.org, even struggling to render a simple Word file correctly.

At the bleeding edges of the software development world some programmers dared to commit heresy; they asked whether the web browser was really the best platform for creating Rich Internet Applications? Looking back they saw a wealth of old desktop software with high fidelity user interfaces and sophisticated interactivity. But this software used old desktop toolkits, bound firmly to one hardware and OS platform. Web pages could be loaded dynamically from the internet on any type of computer; web RIAs were nimble, yet they lacked any capacity for sophistication.

1.1.3 Form follows function: the fall and rebirth of desktop Java

From its first release in 1996 Java had featured a powerful technology for deploying rich applications within a web page. So called *Java Applets* could be placed on any page, just like an image, and ran inside a secure environment which prevented unauthorized tampering

with the underlying operating system. While applets boosted the visibility of the Java brand, the idea initially met with mixed success. The applet was a hard-core programming technology in a world dominated by artists and designers, and while many page authors drooled over Java's power, few understood how to install one onto their own site, let alone how to create an applet from scratch.

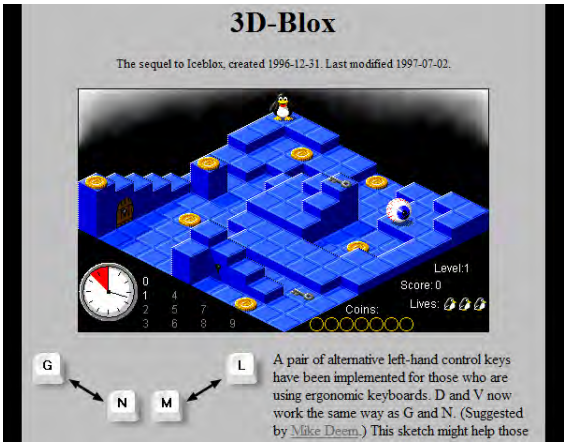


Figure 1.4 An applet (the game 3D-Blox) runs inside a web page, living along side other web content like text and images.

Java applet's main rival was Macromedia Flash, an animation and presentation tool boasting a more designer-friendly development experience. Once Macromedia's plugin began to gain ground the writing was on the wall for the humble Java applet. Already Sun was starting to ignore user-facing Java in favor of big back-end systems running enterprise web applications. The Java applet vanished almost as quickly as it arrived.

Fast forward ten years and the buzz was once again about on-line applications: RIAs and cloud computing. Yet Ajax and HTML were struggling to provide the kind of refined user interaction many now wanted, and Flash's strengths lay more in animation than solid 'functional' GUIs and data manipulation.

Could Java be given a second chance?

Java had proved itself in the enterprise space, amassing many followers in the software community and a vast archive of third party libraries. Yet Java still had one major handicap– on the desktop it remained a tool for cola swigging, black t-shirt wearing, code junkies, not trendy cappuccino sipping, goatee stroking, artists. If Java was to be the answer to the RIA dilemma it needed to be more Leonardo da Vinci, and less Bill Gates.

Figure 1.5 The StudioMoto demo, one of the original JavaFX examples, shows off a glossy user interface with animation, movement and rotating elements all responding to the user's interaction.

In 2005 Sun Microsystems had acquired SeeBeyond Technology Corporation, and in the process picked up a talented software engineer by the name of Chris Oliver. Oliver's experimental F3 programming language (Form Follows Function) sought to make GUI programming easier, by designing the syntax around the specific needs of user interface programming. As they pondered how best to exploit the emerging Rich Internet App market the folks at Sun could surely not have failed to note the potential of combining the existing Java platform with Chris Oliver's new graphics power tool. And so in 2007, at the JavaOne Conference (the community's most important annual gathering), F3 was given center stage as Java's beachhead into the new Rich Internet Application market.

And as if to demonstrate its importance, F3 was blessed with a sexy new name: "*JavaFX*"!

1.2 Minimum effort, maximum impact: a quick shot of JavaFX

It's sometimes hard to visualize the difference a new technology will make to your working life from a description alone. What's often needed is a short but powerful example of what's possible. They say a picture is worth a thousand words, and so in lieu of a few pages of text, I give you figure 1.6.

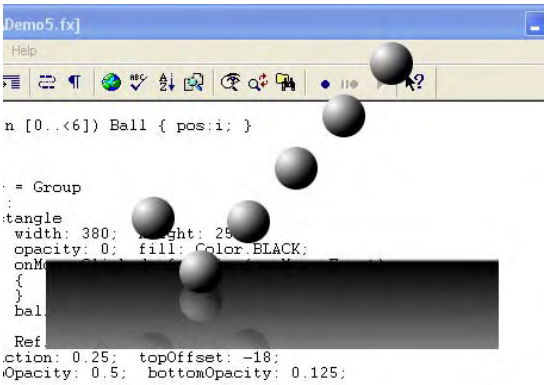


Figure 1.6 The bouncing balls demo, with color shading, reflection effect, and a shaped window (that's a text editor showing underneath, with the source code loaded)

The demo pictured in figure 1.6 is a simple example showing a few of JavaFX's features. The six shaded balls bounce smoothly up and down onto a reflective shaded surface, as the desktop is exposed behind the balls. The window has no title bar (the title bar you see belongs to the text editor behind) but clicking inside its boundary will close the window and exit the bouncing ball application.

Now, the sixty four thousand dollar question: how many lines of code does it take to construct an application like this? Consider what's involved: multiple objects moving independently, circular shading on each ball, linear shading on the ground, a reflection effect, transparency against the desktop, and a click event handler. If you said “less than seventy,” then you'd be right! Indeed, the whole source file is only 1.4k in size, and weighs in at a mere 69 lines.

Don't believe me? Take a look at listing 1.1, below.

Listing 1.1 The bouncing ball demo

```
import javafx.animation.*;
import javafx.stage.*;
import javafx.scene.*;
import javafx.scene.effect.*;
import javafx.scene.shape.*;
import javafx.scene.input.*;
import javafx.scene.paint.*;

var balls = for(i in [0..<6]) {
    var c = Circle {
        translateX: (i*40)+90;
        radius: 18;
        fill: RadialGradient {
```

```
            focusX: 0.25;  focusY:0.25;
            proportional: true;
            stops: [
                Stop { offset: 0; color: Color.WHITE; } ,
                Stop { offset: 1; color: Color.BLACK; }
            ]
        };
    }
}

Stage {
    scene: Scene {
        content: Group {
            content: [
                Rectangle {
                    width: 380;  height: 250;
                    opacity: 0.01;
                    onMouseClicked:
                        function(ev:MouseEvent) { FX.exit(); }
                } , balls
            ]
            effect: Reflection {
                fraction: 0.25;  topOffset: -18;
                topOpacity: 0.5;  bottomOpacity: 0.125;
            }
        }
        fill: LinearGradient {
            endX: 0;  endY: 1;  proportional: true;
            stops: [
                Stop { offset: 0.74;  color: Color.TRANSPARENT; } ,
                Stop { offset: 0.75;  color: Color.BLACK } ,
                Stop { offset: 1;  color: Color.GRAY }
            ]
        }
    };
    style: StageStyle.TRANSPARENT
};

Timeline {
    keyFrames: for(i in [0..<sizeof balls]) {
        KeyFrame {
            time: i*200ms;
            timelines: [
                Timeline {
                    repeatCount: Timeline.INDEFINITE;
                    autoReverse: true;
                    keyFrames: [
                        at (0s) { balls[i].translateY => 30 } ,
                        at (1s) { balls[i].translateY => 230
                            tween Interpolator.EASEIN }
                    ]
                }
            ]
        }
    }
}.play();
```

Since this is an introductory chapter, I'm not going to go into detail about how each part of the code works. Besides, by the time you've finished this book you won't need an explanation of its mysteries, because you'll already be writing cool demos of your own. Suffice to say although the above code may look cryptic now, it's all pretty straight forward once you know the few simple rules which govern the language syntax. Make a mental note, if you want, to check back with listing 1.1 as you read the first half of this book, you'll be surprised at how quickly its secrets are revealed.

1.3 Comparing Java and JavaFX Script: "Hello JavaFX!"

So far we've discussed what JavaFX is, and why it's needed. We've looked at a quick example of JavaFX Script and seen that it's very different to Java, but just how different? For a true side-by-side comparison, to demonstrate the benefits of JavaFX Script over Java, we need to code the same program in both languages. Listings 1.2 and 1.3 do just that.

Listing 1.2 Hello World as JavaFX Script

```
import javafx.scene.Scene;
import javafx.scene.text.*;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
Stage {
   title: "Hello World JavaFX"
    scene: Scene {
        content: Text {
            content:     "Hello World!"
            font:        Font { size: 30 }
            translateX:  114
            translateY:  45
        }
    }
    width:400  height:100
}
```

Listing 1.2 above is a simple JavaFX Script program. Don't panic if you don't understand it yet, this isn't a tutorial, we're merely contrasting the two languages. The program opens a new frame on the desktop with "*Hello World JavaFX*" in the title bar, and the legend "*Hello World!*" as the window contents. Perhaps your untrained eye can already decipher a few clues as to how it works?

Listing 1.3 Hello World as Java

```
import javax.swing.*;
class HelloWorldJava {
    public static void main(String[] args) {
        Runnable r = new Runnable() {
            public void run() {
                JLabel l = new JLabel("Hello World!",JLabel.CENTER);
                l.setFont(l.getFont().deriveFont(30f));
                JFrame f = new JFrame("Hello World Java");
                f.getContentPane().add(l);
                f.setSize(400,100);
```

```
                f.setVisible(true);
            }
        };
        SwingUtilities.invokeLater(r);
    }
}
```

The Java equivalent is presented in listing 1.3 above. It certainly looks busier, although actually it has been stripped back, almost to the point of becoming crude. The Java code is typical of GUIs programmed under popular languages like Java, C++, or BASIC. The frame and the label holding the "*Hello World*" legend are constructed and combined in separate discrete steps. The order of these steps does not necessarily tally with the structure of the user interface they build; the label is created before its parent frame is created, but added after.

Figure 1.7 Separated at birth... “Hello World!” as a JavaFX and Java application.

As the scale of the GUI increases Java's verbose syntax and disjointed structure (compared to the GUI structure) quickly becomes a handful. While JavaFX Script, a bit like the famous Energizer Bunny, can keep on going for far longer thanks to its declarative syntax.

Incidentally, for readers unfamiliar with the Java platform, Appendix D provides a quick overview, including how the “*write once, run anywhere*” promise is achieved, the different editions of Java, and the various versions and revision names over the years. Although JavaFX Script is independent of Java as a language, it's reliance on the Java runtime platform means a little background knowledge of Java is useful.

1.4 Comparing JavaFX with Adobe AIR, GWT and Silverlight.

JavaFX is not the only technology competing to become king of the RIA space: Adobe, Google and Microsoft are all chasing the prize too. But how do their offerings compare to JavaFX? Now we've explored some of the concepts behind JavaFX, we're in a better position to contrast the platform against its alleged rivals.

Comparing technologies is always fraught with danger. Each technology is a multi-faceted beast, and it's impossible to sum all the nuanced arguments up in just a few

paragraphs. Readers are encouraged to seek second opinions if deciding which technology to adopt.

1.4.1 Adobe AIR and Flex

Flex is a toolkit adding application centric features to Flash *movies*, making it easier to write serious web apps, alongside games and animations. AIR (Adobe Integrated Runtime, originally codenamed Apollo) is an attempt to allow Flex web applications to become first class citizens on the desktop. AIR programs can be installed just like regular desktop programs on a PC, Mac or Linux computer, assuming the appropriate AIR runtime has been installed beforehand. Using WebKit (the Open Source HTML/JavaScript component), AIR provides a web-page-like shell into which HTML, JavaScript, Flex, Flash, and PDF content can interact. AIR has made it possible to transfer web programing skills directly onto the desktop, and Adobe plan to extend this concept to allow AIR programmers to target mobile devices as well.

1.4.2 Google Web Toolkit

Google Web Toolkit is an Open Source attempt to smooth over the bumps in HTML/Ajax application development with a consistent cross-browser JavaScript library of desktop-inspired widgets and functions. It's said GWT started as an internal Google project to help write sites like GMail and Google Calendar (although which Google sites actually use GWT is unknown). GWT applications are coded in Java, compiled to JavaScript, and run entirely within the web browser. They can make use of optionally installed plug-ins, such as *Gears*, to provide off-line support.

1.4.3 Microsoft Silverlight

With Silverlight, Microsoft is seeking to shift its desktop software domination to inside the browser. Silverlight is a proprietary browser plug-in for recent editions of Windows and Mac OS X. Linux is also covered via an Open Source project and a deal with Novell (licensing difficulties may exist for non-Novell Linux customers.) Silverlight supports rich vector-based user interfaces, coded in .NET languages (like C#) and a UI markup language called XAML (Extensible Application Markup Language.) Microsoft worked hard to create a fluid video/multimedia environment, with solid support for all its Windows Media formats.

1.4.4 And by comparison... JavaFX

While other RIA technologies blur the line between desktop and browser, JavaFX removes the distinction entirely. A single JavaFX application can move seamlessly (quite literally, by being dragged from the browser window) from one environment to the other. Smart phones and TV boxes will join this list at a later date. With a common core across all environments, complimented by device-specific extensions, JavaFX lets us target every device, or exploit the full power of a particular device.

While other RIA technologies recycle existing languages, JavaFX Script is built from the ground up specifically for creating sophisticated user interfaces and animation. Studying

common working methods found in UI software, the JavaFX team created a language around those patterns. The declarative syntax permits code and structure to be interwoven with a degree of ease not found in the schizophrenic bi-lingual approach of its rivals. Binding defines a direct relationship between an object and the data or functions it depends upon; the heavy lifting of Model/View/Controller is done for you. And because JavaFX Script is compatible with Java classes, it has access to over a decade of libraries and Open Source projects.

It's true the need to learn a new language may discourage some, but the reward is a much more powerful tool, shaped specifically for the job at hand. Picking the best tool can often mean the difference between success and failure, while holding on to our familiar tools for too long can sometimes put us at a disadvantage. The skill is in knowing when to embrace a new technology, and hopefully this section has helped clarify whether JavaFX is the right technology for you!

1.5 "But why should I buy THIS book?"

Good question, indeed why a book at all? The APIs are documented on-line, and there are blogs aplenty guiding coders through that tricky first application.

This book specifically seeks not to regurgitate existing documentation, like so many programming tomes tend to. For that reason you won't find laborious enumerations of every variation of every shade of every nuance of every class. This book assumes you're intelligent enough to read the documentation for yourself, once pointed in the right direction; you don't need it reprinted here. So what *is* in this book?

The early chapters give a quick and entertaining (yet comprehensive) guide to the JavaFX Script language, then it's straight into the projects! Each project chapter houses a self contained mini-application requiring specific skills, and works from initial goals, towards a solution in JavaFX. Successive projects reinforce acquired skills, and add new ones. Concepts are demonstrated and explained in real world scenarios; it's an approach centered around common practices, solutions and patterns, rather than merely ticking off every variation of (for example) a scene graph node or animated transition included in the API.

The code in each chapter seeks to be ideas-rich, but compact and fresh. What's the point of page upon page of stuff the reader already saw last chapter? Although functional, each completed project leaves room for readers to experiment further, practicing new found skills by adding features, or polishing the UI with extra color blends and animations.

For better or worse, the text attempts to remain agnostic of any particular IDE or tool, other than those shipped with the standard JavaFX SDK. Illustrated click-by-click guides for each IDE would be page hungry, and offer little over the on-line tutorials already provided with (or for) each plug-in. Again, it's about complimenting available documentation, not reproducing it; more room for JavaFX examples and advice, not IDE-specific tutorials (this is, after all, *JavaFX in Action* not *NetBeans in Action*!) Relevant plug-in/IDE links are provided in the appendices.

So, is this book for you? If you're merely looking for a hard copy of the API documentation, then perhaps not. But if you want something which goes a little deeper, exploring JavaFX through *real world* code, solving *real world* problems, then I hope you'll find what you're looking for in the pages to come.

1.6 Summary

This chapter has been an introduction to the world of JavaFX and JavaFX Script. We started by considering the power of Domain Specific Languages, designed specifically to meet the needs of particular tasks. Then we considered the rise of the Rich Internet Application and the challenges in developing them using current browser-based technologies. We revisited Java's disappointing track record on the desktop, particularly with lightweight internet applications like applet, but saw how this could change with the introduction of JavaFX to address a new generation of internet applications. We saw an example of JavaFX Script doing modestly impressive things in only a few dozen lines of code, and reviewed side-by-side the differences in styles and size of Java and JavaFX Script source code. Then, finally, we considered how JavaFX stacks up against the apparent opposition.

I hope this has been enough to grab your attention, and fire your imagination, because in the very next chapter we leave the theory behind and dive straight into the detail.

Over the next couple of chapters we'll tour the JavaFX Script language, with I hope plenty of nice surprises along the way. This will get us ready to tackle subsequent chapters, where we use practical mini-projects to demonstrate different aspects of JavaFX. (For those expert Java programmers who would prefer more of a whistle stop tour of the new language, Appendix B acts as both a *flash card* tutorial, and an aide-mémoire).

Before we move on you will almost certainly want to take a detour to Appendix A, which acts as a setup guide for downloading and installing JavaFX, plus getting your code to build. It also features some very useful JavaFX links for help and further reading. Also, if you're not a Java programmer, can I remind you of the introduction to the Java platform (and how JavaFX fits into it) in Appendix D.

So that's the introduction out of the way. Are you excited? Well, I certainly hope so! Let the fun begin...

2

JavaFX Script: data and variables

If the previous chapter has had the desired effect you should be eager to get your hands dirty with some code. But before we can start dazzling unsuspecting bystanders with our stunning JFX visuals we'll need to get acquainted with the JavaFX's own programming language, *JavaFX Script*. In this chapter, and the one which follows, we'll start to do just that.

In this chapter we will look at how variables are created and manipulated. JavaFX Script has a lot of interesting features in this area, beyond those offered by languages like Java or JavaScript, such as sophisticated array manipulation and the ability to bind variables into automatic update relationships. By the end of this chapter you'll understand how these features work, and next chapter we'll build on this knowledge, exploring how they integrate into standard programming constructs like loops, conditions and classes.

In writing this tutorial I've attempted to create a smooth progression through the language, with later sections building on the knowledge gleaned from previous reading. However, this logical progression wasn't always possible, and just occasionally later detail bleeds a little into earlier sections.

QUICK START

If you don't fancy a full-on tutorial right now, and you consider yourself a good enough Java programmer, you might try picking up the basics of JavaFX Script from the quick reference guide in Appendix B.

Each of the code snippets in this tutorial should be runnable as they are. Many output to 'standard out', and when this is the case the console output is presented following the code snippet, in bold text.

One final note: from now on I'll occasionally resort to the familiar terms “JavaFX” or “JFX” in lieu of the more formal language title. Strictly speaking this is wrong (JavaFX is the

platform and not the language) but you'll forgive me if I err on the side of making the prose more digestible, at the risk of annoying a few pedants (you know who you are!)

2.1 Annotating code with comments

Before we begin in earnest, let's take a look at JavaFX Script's method for code commenting. That way you'll be able to comment your own code as you *play along at home* during the sections to come. Actually, there's not a lot to cover, because JavaFX Script uses the same C++ like comment syntax as Java and many other popular languages.

Listing 2.1 JavaFX Script comments

```
// A single line comment.

/* A multi line comment.
   Continuing on this line.
   And this one too! */

/* Another multi line comment
 * in a style much preferred by
 * many programmers...
 */
```

That was short and sweet. Unlike listing 2.1, the source code in this book is devoid of inline comments, to keep the examples tight on the printed page (reducing the need to flip to and from multiple pages when studying a listing), but as you experiment with your own code, I strongly recommend you use comments to annotate it. Not only is this good practice, but it helps your *little gray cells* reinforce your newly acquired knowledge.

Give reasons for your answer

There's undoubtedly a certain resistance to source code commenting amongst programmers. It was ever thus! Comments acquired a bad name during the 1970s and 80s, when the prevailing mood was for vast quantities of documentation about every variable and parameter. Pure overkill. I'd suggest the ideal use for comments is in explaining the reasoning behind an algorithm. "*Give reasons for your answer*", as countless high school examination papers would demand. To the oft heard complaint "my code is self documenting", I'd counter "only to the compiler!" Justifying your ideas in a line or two before each code "paragraph" is a useful discipline for double-checking your own thinking, with a bonus that it helps fellow coders spot when your code doesn't live up to the promise of your comments.

Now we know how to write code the compiler ignores, let's move on to write something which has an effect. We'll begin with basic data types.

2.2 Data types

At the heart of any language is data and data manipulation. Numbers, conditions, and text are all typical candidates for data types, and indeed JavaFX Script has types to represent all

of these. But being a language centered around animation, it also features a type to represent time.

JavaFX Script's approach to variables is slightly different to Java. Whereas Java employs a dual strategy, respecting both the *high level* objects of Object Orientation and *low level* types of Java bytecode, JavaFX Script only has objects. That's not to say it doesn't have any of the syntactic conveniences of Java's primitive types, as we'll see in the following sections.

2.2.1 Static, not dynamic types

Variables in JavaFX Script are statically typed, meaning each variable holds a given type of information, which allows only a compatible range of operations to be performed upon it. Strings will not, for example, magically turn themselves into numbers so we can perform arithmetic on them, even if said strings contains only valid number characters. In that regard they work the same way as the Java language.

Java novices, or other curious souls, can consult section C1 of Appendix C for more on static versus dynamic variable types in programming languages.

2.2.2 Value type declaration (var, Boolean, Integer, Number, String)

Value types are the core building blocks for data in JavaFX Script, designed to hold commonplace data like numbers and text. Unlike Java primitives, JavaFX Script's value types are fully fledged objects, with all the added goodness which stems from using classes.

One thing making value types stand out from their object brethren is special provision is made for them in the JavaFX Script syntax. Not sure what this means? Consider the humble String class (`java.lang.String`) in Java: although just another class, `String` is blessed with its own syntax variants for creation and concatenation, without need of constructors and methods. This is just syntactic sugar, behind the scenes the syntax is replaced by the familiar constructors and methods, but the string syntax keeps the source code readable.

Another difference between value types and other objects is uninitialized (unassigned) value types assume a default value rather than `null`. Indeed value types cannot be assigned a null value. We'll look at default values shortly.

JavaFX Script offers several basic types, detailed in table 2.1 below.

Table2.1 JavaFX Script basic types

Type	Details	Java equivalent
Boolean	True or false flag	`java.lang.Boolean`
Byte	Signed 8 bit integer. JFX 1.1+	`java.lang.Byte`
Character	Unsigned 16 bit Unicode. JFX 1.1+	`java.lang.Character`
Double	Signed 64 bit fraction. JFX 1.1+	`java.lang.Double`
Duration	Time interval	None

Type	Details	Java equivalent
Float	Signed 32 bit fraction. JFX 1.1+	`java.lang.Float`
Integer	Signed 32 bit integer	`java.lang.Integer`
Long	Signed 64 bit integer. JFX 1.1+	`java.lang.Long`
Number	Signed 32 bit fraction.	`java.lang.Float`
Short	Signed 16 bit integer. JFX 1.1+	`java.lang.Short`
String	Unicode text string	`java.lang.String`

In this section we'll focus on the first four, which are recognizable as variations on the basic primitive types you often find in other programming languages. The `Duration` type is JavaFX Script specific, reflecting its focus on animation. You can look forward to getting better acquainted with `Duration` later in this chapter.

Changes for JavaFX Script 1.1

In JavaFX Script 1.0 the only fractional (floating point) type was `Number`, which had double precision, making it equivalent to Java's `Double`. However in JavaFX Script 1.1 six new types were introduced: `Byte`, `Character`, `Double`, `Float`, `Long` and `Short`. This resulted in `Number` being downsized to 32 bit precision, effectively becoming an alternative name for the new `Float` type.

Let's look at some example code (listing 2.2) defining the four types we're looking at in this section:

Listing 2.2 Defining value types

```
var valBool:Boolean = true;
var valByte:Byte = -123;
var valChar:Character = 65;
var valDouble:Double = 1.23456789;
var valFloat:Float = 1.23456789;
var valInt:Integer = 8;
var valLong:Long = 123456789;
var valNum:Number = 1.245;
var valShort:Short = 1234;
var valStr:String = "Example text";
println("Assigned: B: {valBool}, By: {valByte}, "
    "C: {valChar}, D: {valDouble}, F: {valFloat}, ");
println("  I: {valInt}, L: {valLong}, N: {valNum}, "
    "Sh: {valShort}, S: {valStr}");

var hexInt:Integer = 0x20;
var octInt:Integer = 040;
var eNum:Double = 1.234E-56;
println("hexInt: {hexInt}, octInt: {octInt}, "
     "eNum: {eNum}");
```

```
Assigned: B: true, By: -123, C: A, D: 1.23456789, F: 1.2345679,
  I: 8, L: 123456789, N: 1.245, Sh: 1234, S: Example text
hexInt: 32, octInt: 32, eNum: 1.234E-56
```

The keyword `var` begins variable declarations, followed by the variable name itself. Next comes a colon and the variable's type, and after this an equals symbol and the variable's initial value. A semi-colon is used to close the declaration.

Keen eyed readers will have spotted the use of the keyword `true` in the boolean declaration; just as in Java, `true` and `false` are reserved words in JavaFX Script. You may also have noted the differing results for the `Float` and `Double` types, born out of their differences in precision.

It's also possible to express integers using hexadecimal or octal, and floating point values using scientific E notation, as per other programming languages. The final output line shows this in effect.

println() and strange looking strings

The listings in this section, and other sections in this chapter, make reference to a certain `println()` function. Common to all JavaFX objects, thanks to its inclusion in the `javafx.lang.FX` base class, `println()`. is the JavaFX way of writing to the application's default output stream. Java programmers will be familiar with `println()` through Java's `System.out` object, but may not recognize the bizarre curly braced strings being used to create the formatted output. For now just accept them–we'll deal with the details in a few pages time.

As in many languages, the initializer is optional. We could have ended each declaration after its type, with just closing semi-colons, resulting in each variable being initialized to a sensible default value.

Listing 2.3 Defining value types using defaults

```
var defBool:Boolean;
var defInt:Integer;
var defNum:Number;
var defStr:String;
println("Default: B: {defBool}, "
    "I: {defInt}, N: {defNum}, S: {defStr}");
```

```
Default: B: false, I: 0, N: 0.0, S:
```

You'll note in listing 2.3 the absence of any initial values. Despite being objects, value types cannot be `null`, so defaults of false, zero or empty are used. But the initial value is not the only thing we can leave off, as the next snippet shows:

Listing 2.4 Defining value types using type inference

```
var infBool = true;
var infFlt = 1.0;                                                    A
var infInt = 1;
```

```
var infStr = "Some text";
println("Infered: B: {infBool}, "
    "F: {infFlt}, I: {infInt}, S: {infStr}");
println("{infFlt.getClass()}");
```

```
Infered: B: true, I: 1, N: 1.1, S: Some text
class java.lang.Float
```

A Be careful declaring real numbers this way

JavaFX Script supports *type inference* when declaring variables. In plain English this means if an initializer is used and the compiler can unambiguously deduce the variable's type from it, you can omit the explicit type declaration. In listing 2.4, if we'd initialized `assFlt` with just the value 1 it would have become an `Integer` instead of a `Float`, quoting the fractional part drops a hint as to our intended type.

2.2.3 Initialize-only and reassignable variables (var, def)

So far we've see variables declared using the `var` keyword. But there is a second way of declaring variables, as listing 2.5 is about to reveal.

Listing 2.5 Declaring variables with def

```
var canAssign:Integer = 5;
def cannotAssign:Integer = 5;
canAssign = 55;
//cannotAssign = 55;
A
```

A Compiler error if uncommented

Using `def` instead of `var` results in variables which cannot be re-assigned once created.

Don't be tempted to assume `def` creates *old fashioned* constants, like Java's `final` keyword. When assigned simple values (like listing 2.5) a `def` variable is indeed a constant, but in other circumstances a `def` variable's contents **can and do** change. In a later section we'll investigate bound variables, revisiting `def` to see how a variable can change its value, without actually changing its content.

So, ignoring bound variables for the moment, a valid question is "when should we use `var` and when should we use `def`?" Well, firstly, `def` is useful if we want to drop hints to fellow programmers as to how a given variable should be used; secondly, the compiler can detect misuse of a variable if it knows how we intend to use it; but crucially, JFX can better optimize our software if given extra information about the data its working with.

For simple assignments like those in listing 2.5, it's largely a matter of choice or style. Using `def` helps make our intentions clear, and means our code might run a shade faster.

2.2.4 Arithmetic on value types (+, -, etc.)

Waxing lyrical about JavaFX Script's arithmetic capabilities is pointless: they're basically the same as most programming languages. Unlike Java's, however, JavaFX Script's value types are objects, so they respond to both conventional operator syntax (like Java primitives), and method calls (like Java objects.) Let's see how that works in practice:

Listing 2.6 Arithmetic on value types

```
def n1:Number = 1.5;
def n2:Number = 2.0;
var nAdd = n1 + n2;
var nSub = n1 - n2;
var nMul = n1 * n2;
var nDiv = n1 / n2;
var iNum = n1.intValue();                                          A
println("nAdd = {nAdd}, nSub = {nSub}, "
    "nMul = {nMul}, nDiv = {nDiv}");
println("iNum = {iNum}");
```

```
nAdd = 3.5, nSub = -0.5, nMul = 3.0, nDiv = 0.75
iNum = 1
```

A Converting n1 to an integer

The variables `n1` and `n2` (listing 2.6) are initialized and a handful of basic mathematical operations are performed. Note the conversion of `n1` to an integer via `intValue()`, made possible by value types actually being objects. We'll look at objects a little later–for now be aware value types are more than just primitives.

Table 2.2 List of arithmetic operators

Operator	Function
+	Addition
-	subtraction; unary negation
*	Multiplication
/	Division
mod	Remainder (Java's equivalent is %)
+=	Addition and assign
-=	Subtract and assign
*=	Multiply and assign
/=	Divide and assign
++	Prefix (pre-increment assign) / suffix (post-increment assign)
--	Prefix (pre-decrement assign) / suffix (post-decrement assign)

In table 2.2, above, we've a list of the common arithmetic operators; let's see some of them in action:

Listing 2.7 Further examples of arithmetic operations

```
var int1 = 10;
var int2 = 10;
int1 *= 2;
int2 *= 5;
```

```
var int3 = 9 mod (4+2*2);
var num:Number = 1.0/(2.5).intValue();                                    A
println("int1 = {int1}, "
    "int2 = {int2}, int3 = {int3}, num = {num}");

int1 = 20, int2 = 50, int3 = 1, num = 0.5
 A  Yes, even number literals are objects
```

It's all very familiar. The only oddity in listing 2.7 is the value 2.5 having a method called upon it (surrounding brackets avoid ambiguity over the 'point' character.) This is yet another example of the lack of true primitives–including literals too! The numeric literal 2.5 became a JavaFX Number type, allowing function calls on it.

2.2.5 Logic operators (and, or, <, >, =, >=, <=, !=)

In the next chapter we'll look at code constructs like conditions. But, as we're currently exploring the topic of data we might as well take a quick gander at how JavaFX implements the logic operations which form the backbone of condition code.

Listing 2.8 Logic operators

```
def testVal = 99;
var flag1 = (testVal == 99);
var flag2 = (testVal != 99);
var flag3 = (testVal <= 100);
var flag4 = (flag1 or flag2);
var flag5 = (flag1 and flag2);
var today:java.util.Date = new java.util.Date();                          A
var flag6 = (today instanceof java.util.Date);
println("flag1 = {flag1}, flag2 = {flag2}, "
     "flag3 = {flag3}");
println("flag4 = {flag4}, flag5 = {flag5}, "
     "flag6 = {flag6}");

flag1 = true, flag2 = false, flag3 = true
flag4 = true, flag5 = false, flag6 = true
 A  Creating a Java Date object
```

For most programmers, apart from a few differences in keywords (eg. `and` and `or` instead of && and `||`), there's no great surprises lurking in listing 2.8. As in Java, the `instanceof` operator is used for testing the type of an object, in this case a Java `Date`. It's all rather predictable–but then, isn't that a good thing?

2.2.6 Translating and checking types (as, instanceof)

Because JavaFX Script is a statically typed language, casts may be required to concert one data type into another. If the conversion is guaranteed to be safe, JavaFX Script just allows it without any fuss, but where there's potential for error or ambiguity JavaFX Script insists on a cast.

Casts are performed by appending the keyword `as`, followed by the desired type, after the variable or term which needs converting.

Listing 2.9 Casting types

```
import java.lang.System;
var pseudoRnd:Integer =
    (System.currentTimeMillis() as Integer) mod 1000;

var str:java.lang.Object = "A string";
var inst1 = (str instanceof String);
var inst2 = (str instanceof java.lang.Boolean);
println("String={inst1} Boolean={inst2}");
```

```
String=true Boolean=false
```

Usually casting is required when a larger object is downsized to a small one, as in the example above (listing 2.9), where a 64 bit value is being truncated to a 32 bit value. Another example is when an object type is being changed *down* the hierarchy of classes, to a more specific subclass. For non-Java programmers there's a little more detail on the purpose of casts in Appendix C–look under section C2.

Checking the type of a variable can be done with `instanceof`, which returns `true` of the variable matches the supplied type, and `false` if it doesn't.

This concludes our look at basic data types. If it's been all rather tame for you, don't worry, the next section will start to introduce some JFX power tools.

2.3 Working with text, via strings

Where would we be without strings? This book wouldn't be getting written, that's for sure (pounding the whole manuscript out on a manual typewriter somehow doesn't endear itself.) We looked at defining basic string variables previously, now let's delve deeper to unlock the power of string manipulation in JavaFX.

2.3.1 String literals and embedded expressions

String literals allow us to write strings directly into the source code of our programs. JavaFX Script string literals can be defined using single or double quotes, as the following code (listing 2.10) demonstrates.

Listing 2.10 Basic string definitions

```
def str1 = 'Single quotes';
def str2 = "Double quotes";
println("str1 = {str1}");
println("str2 = {str2}");
```

```
str1 = Single quotes
str2 = Double quotes
```

Is there a difference between these two styles? No, either double or single quotes can be used to enclose a string, and providing both ends of the string match it really doesn't matter which you use.

Listing 2.11 Multi line strings, double and single quoted strings

```
def multiline = "This string starts here, "
'and ends here!';
println("multiline = {multiline}");
```

```
println("UK authors prefer 'single quotes'");
println('US authors prefer "double quotes"');
println('I use "US" and \'UK\' quotes');
```

```
multiline = This string starts here, and ends here!
UK authors prefer 'single quotes'
US authors prefer "double quotes"
I use "US" and 'UK' quotes
```

String literals next to each other in the source code, with nothing in between (except whitespace), are concatenated together. Even if they are separated by a new line, as demonstrated by listing 2.11 above.

Single quoted strings can contain double quote characters, and double quoted strings can contain single quote characters, but to use the same quote as delimits the string you need to escape it with a backslash. This ability to switch quote styles is particularly useful when working with other languages from within JavaFX Script, like SQL. (By the way: yes, I know many *modern* British novelists prefer double quotes!)

In the listings we've seen previously some of the strings have contained a strange curly brace format for incorporating variables. Now it's time to find out what that's all about.

Listing 2.12 Strings with embedded expressions

```
def rating = "cool";
def eval1 = "JavaFX is {rating}!";
def eval2 = "JavaFX is \{rating\}!";
println("eval1 = {eval1}");
println("eval2 = {eval2}");

def flag = true;
def eval3 =
    "JavaFX is {if(flag) "cool" else "uncool"}!";
println("eval3 = {eval3}");
```

```
eval1 = JavaFX is cool!
eval2 = JavaFX is {rating}!
eval3 = JavaFX is cool!
```

Strings can contain expressions, enclosed in curly braces. When encountered, the expression body is executed, and the result substituted for the expression itself. In listing 2.12 we see the value of the variable, `rating`, is inserted into the string content of `eval1` by enclosing a reference to the `rating` in braces. Escaping the braces with backslashes prevents any attempted expression evaluation, as has happened with `evall2`.

We can get more adventurous than just simple variable references. The condition embedded inside the curly braces of `eval3` displays either "JavaFX is cool!" or "JavaFX is uncool!", depending upon the contents of the boolean variable `flag`. We'll be looking at the if/else syntax later, of course, but for now let's continue our exploration of strings, because JavaFX Script offers even more devious ways to manipulate their contents.

2.3.2 String formatting

We've seen how to expand a variable into a string using the curly brace syntax, but this is only the tip of the iceberg. Java has a handy class called `java.util.Formatter` which permits string control similar to C's infamous `printf()` function. The `Formatter` class allows commonly displayed data types, specifically strings, numbers and dates, to be translated into text form based on a pattern.

Listing 2.13 String formatting

```
import java.util.Calendar;
import java.util.Date;

def magic = -889275714;
println("{magic} in hex is {%08x magic}");                                A

def cal:Calendar = Calendar.getInstance();
cal.set(1991,2,4);
def joesBirthday:Date = cal.getTime();
println("Joe was born on a {%tA joesBirthday}");                          B
```

```
-889275714 in hex is cafebabe
Joe was born on a Monday
 A Eight digit hexadecimal
 B Display date's week day
```

In the two `println()` calls of listing 2.13 we see the string formatter in action. The first uses a format of `"%08x"` to display an eight digit hexadecimal represent ion of the value of `magic`. The second example creates a date for 4th March 1991 using Java's `Calendar` and `Date` classes. The format pattern `"%tA"` displays the day name (Monday, Tuesday...) from any date it is applied to.

For more details on the various formatting options, consult the Java documentation for the `java.util.Formatter` class.

Being negative

Perhaps you're wondering how the hexadecimal value CAFEBABE could be represented in decimal as -889275714, not 3405691582? Like Java, JFX uses 32 bit signed integers, meaning the lower 31 binary digits of each integer represent its value and the most significant digit stores whether the value is positive or negative. As the hex value CAFEBABE uses all 32 bits, its 32nd bit causes JFX to see it as negative in many circumstances. If we tried to use the number 3405691582 in the source code the compiler would inform us it's too big to store in an integer. However -889275714 results in exactly the same 32 bit pattern.

In case you didn't know: CAFEBABE is the four byte 'magic' identifier starting every valid Java bytecode class file. You learn something new every day! (Or maybe not!)

As well as in-built string formatting, JavaFX Script also has a specific syntax for string localization, which is what we'll look at next.

2.3.3 String localization

The internet is a global phenomenon, and while it might save us programmers a whole load of pain if everyone would just agree on a single language, calendar and daylight saving time, the fact is people cherish their local culture and customs and our software should respect that. An application might use dozens, hundreds, or even thousands of bits of text to communicate to the user. For true internationalization these need to be changeable at runtime to reflect the native language of the user (or, at least, the language settings of the computer the user is using).

To do this we use individual property files, each detailing the strings to be used for a given language.

Listing 2.14 String localization: the <classname>_en_UK.fxproperties file

```
"Trash" = "Rubbish"
"STR_KEY" = "UK version"
```

Listing 2.14 is a simple two line localization property file for UK English. It's filename and location is important.

The filename must begin with the name of the script it is used in, followed by an underscore character, then a valid ISO Language Code as specified in the standard, ISO-639.2. These codes comprise two lower case characters, signifying which language the localization file should be used for. If a specific variant of a language is required (for example, UK English instead of US English) a further underscore and an ISO Country Code may be added, as specified by ISO-3166. These codes are two upper case characters, documenting a specific country. Finally the extension "`.fxproperties`" must be added.

In the book's source code the script for testing the above is called `Examples3.fx`, so the properties file for UK English would be `Examples3_en_UK.fxproperties`. If we created a companion property file for all Japanese regions it would be called `Examples3_ja.fxproperties` ("ja" being the language code for Japanese).

All properties files must be placed somewhere on the classpath, so JavaFX can find them at run time. Now let's look at the code to test our localization strings.

Listing 2.15 String localization

```
println(##"Trash");
println(##[STR_KEY]"Default version");

Trash
Default version

Rubbish
UK version
```

Listing 2.15 shows two examples of localized strings in JavaFX Script, and the output for two runs of the program, one non-UK and the other UK.

The ## prefix means JFX will look for an appropriate localization file, and use its contents to replace the following string. In the first line of code the string "`Trash`" is replaced by "`Bin`" using the `_en_UK` file we created earlier. The "`Trash`" string is used as a key to look

up the replacement in the properties file. If we're not running the program in the United Kingdom, or the property file cannot be found/loaded for some reason, the "`Trash`" string is used as a default.

The second line does exactly the same, except the key and the default are separate. So "STR_KEY" is used as a key to look up the localization property, and "`Default version`" is used if no suitable replacement can be found.

You might be wondering how you can test the code without bloating your carbon footprint with a round-the-World trip. Fortunately the JVM has a couple of system properties to control its region and language settings, namely `user.region` and `user.language`. They should be settable within the testing environment of your IDE (look under "System properties"), or from the command line using the `-D` switch. Here's a couple of example:

```
-Duser.region=UK -Duser.language=en
-Duser.region=US
```

And that's strings. Thus far we've looked at very familiar data types, but next we'll look at a value type you won't find in any other popular programming language: the duration.

2.4 Durations, using time literals

You *really* know a language is specialized to cope with animation when you see it has a literal syntax for time durations. What are *time literals*? Well, just as the quoted syntax makes creating strings easy, the time literal syntax provides a simple, specialized, notation for expressing intervals of time.

Listing 2.16 Declaring duration types

```
def mil = 25ms;                                                    A
def sec = 30s;                                                     A
def min = 15m;                                                     A
def hrs = 2h;                                                      A
println("mil = {mil}, sec = {sec}, "
    "min = {min}, hrs = {hrs}");                                   B
```

```
mil = 25ms, sec = 30000ms, min = 900000ms, hrs = 7200000ms
```

A Milliseconds, seconds, minutes and hours
B Output is milliseconds

Appending 'ms', 's', 'm' or 'h' to a value creates an object of type `Duration` (not unlike wrapping characters with quotes turns them into a `String`.) Listing 2.16 demonstrates various times expressed using the literal notation: 25 milliseconds, 30 seconds, 15 minutes and 2 hours. (Note: no matter how they were defined, `Duration` objects default to milliseconds when `toString()` is called on them.) This handy notation is pretty versatile, and can be used in a variety of ways, including with arithmetic and boolean logic.

Listing 2.17 Arithmetic on duration types

```
import java.lang.System;
def dur1 = 15m * 4;                                                A
```

```
def dur2 = 0.5h * 2;                                                    B
def flag1 = (dur1 == dur2);
def flag2 = (dur1 > 55m);
def flag3 = (dur2 < 123ms);
System.out.printf(
    "dur1 = {dur1.toMinutes()}m , "                                     C
    "dur2 = {dur2.toMinutes()}m%n"                                      C
    "(dur1 == dur2) is {flag1}%n"
    "(dur1 > 55m) is {flag2}%n"
    "(dur1 < 123ms) is {flag3}%n");
```

```
dur1 = 60.0m , dur2 = 60.0m
(dur1 == dur2) is true
(dur1 > 55m) is true
(dur1 < 123ms) is false
```

A **15 minutes times 4**
B **Half an hour times 2**
C **Converted to minutes**

In listing 2.17, both `dur1` and `dur2` are created as `Duration` objects, set to one hour. The first is created by multiplying fifteen minutes by four, and the second is created by multiplying half an hour by two. These `Duration` objects are then compared against each other, and against other literal `Duration` objects. To make the console output more readable we convert the usual millisecond representation to minutes.

When we start playing with animation we'll see how time literals help create compact, readable, source code for all manner of visual effects. But that's after we've mastered JavaFX Script. Next up is sequences.

2.5 Sequences: not quite arrays

Sequences are collections of objects or values with a logical ordered relationship. As the saying goes, they do “exactly what it says on the tin”; in other words, a sequence is a sequence of *things*!

It's tempting to think of sequences as arrays by another name, indeed they can be used for array-like functionality, but sequences hide some pretty clever extra functionality making them more useful to the type of work JavaFX is designed to do. In the following sections we'll look at how to define, extend, retract, slice and filter JavaFX sequences. Sequences have quite a rich syntax associated with them, so let's jump straight in.

2.5.1 Basic sequence declaration and access (sizeof)

We won't get very far if we cannot define new sequences. The following code snippet shows us how to do just that:

Listing 2.18 Sequence declaration

```
import java.lang.System;
def seq1:String[] = [ "A" , "B" , "C" ];
def seq2:String[] = [ seq1 , "D" , "E" ];
def flag1 = (seq2 == ["A","B","C","D","E"]);
def size = sizeof seq1;
System.out.printf("seq1 = {seq1.toString()}\n"
```

```
    "seq2 = {seq2.toString()}\n"
    "flag1 = {flag1}\n"
    "size = {size}\n");
```

```
seq1 = [ A, B, C ]
seq2 = [ A, B, C, D, E ]
flag1 = true
size = 3
```

Listing 2.18 exposes a few important sequence concepts:

- A new sequence is declared using a square brackets syntax.
- When one sequence is used inside another, it is expanded in place.
- Sequences are equal if they are the same size and each corresponding element is equal.
- The notation for referring to the type of a sequence uses square brackets after the plain object type. For example `String[]` refers to a sequence of `String` objects.
- The `sizeof` operator can be used to determine the length of a sequence.

To reference a value in a sequence we use the same square bracket syntax as many other programming languages. The first element is at index zero, as listing 2.19 proves.

Listing 2.19 Referencing a sequence element

```
import java.lang.System;
def faceCards = [ "Jack" , "Queen" , "King" ];
var king = faceCards[2];
def ints = [10,11,12,13,14,15,16,17,18,19,20];
var oneInt = ints[3];
var outside = ints[-1];
System.out.printf("faceCards[2] = {king}\n"
    "ints[3] = {oneInt}\n"
    "ints[-1] = {outside}\n");
```

```
faceCards[2] = King
ints[3] = 13
ints[-1] = 0
```

You'll note how referring to an element outside of the sequence bounds returns a default value, rather than an error or an exception. Apart from this oddity the code looks remarkably close to arrays in other programming languages–so, what about those clever features I promised? Let's experience our first bit of sequence cleverness by looking at ranges and slices.

2.5.2 Sequence creation using ranges ([..], step)

The examples thus far have seen sequences created explicitly, with content as comma separated lists inside square brackets. This may not always be convenient, so JFX supports a handy range syntax.

Listing 2.20 Sequence creation using a range

```
def seq3 = [ 1 .. 100 ];                                                A
println("seq3[0] = {seq3[0]},"
    "seq3[11] = {seq3[11]}, seq3[89] = {seq3[89]}");
```

```
seq3[0] = 1,seq3[11] = 12, seq3[89] = 90
 A  Two dots creates a range
```

The sequence in listing 2.20 is populated with the values 1 to 100, inclusive. Two dots separate the start and end delimiters of the range, enclosed in the familiar square brackets. Is that all there is to it? No, not by a long stretch!

Listing 2.21 Sequence creation using a stepped range

```
def range1 = [0..100 step 5];
def range2 = [100..0 step -5];
def range3 = [0..100 step -5];
def range4 = [0.0 .. 1.0 step 0.25];
println("range1 = {range1.toString()}");
println("range2 = {range2.toString()}");
println("range3 = {range3.toString()}");
println("range4 = {range4.toString()}");
```

```
range1 = [ 0, 5, 10, 15, 20, 25, 30, 35, 40, 45,
[CA]50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100 ]
range2 = [ 100, 95, 90, 85, 80, 75, 70, 65, 60, 55,
[CA]50, 45, 40, 35, 30, 25, 20, 15, 10, 5, 0 ]
range3 = [ ]
range4 = [ 0.0, 0.25, 0.5, 0.75, 1.0 ]
```

The above cyan [CA]'s are code continuations, for wrapped lines.

Listing 2.21 shows various ranges created using an extra step parameter. The first runs from zero to one hundred in steps of five (0, 5, 10 ..) and the second does the same in reverse (100, 95, 90 ..) The third goes from zero to one hundred backwards, resulting (quite rightly!) in an empty sequence. Finally, just to prove ranges aren't all about integers, we have a range of type `Number[]` from zero to one in steps of a quarter.

We can include ranges inside larger declarations. You'll recall we can expand one sequence inside another, and we can exploit this fact for more readable source code.

Listing 2.22 Expanding one sequence inside another

```
def blackjackValues = [ [1..10] , 10,10,10 ];                           A
println("blackjackValues = "
    "{blackjackValues.toString()}");
```

```
blackjackValues = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10 ]
 A  Expanding a range inside another declaration
```

The example above (listing 2.22) creates a sequence representing the card values in the game of Blackjack: aces to tens use their face value, while picture cards (jack, queen, king) are valued at ten. (Yes, *I am* aware aces are also eleven–what do you want, blood?!)

2.5.3 Sequence creation using slices ([..<])

The range syntax can be useful in many circumstances, but it's not the only trick JavaFX Script has up its sleeve. For situations which demand a little more control, we can also take a slice from an existing sequence to create a new one, as listing 2.23 shows:

Listing 2.23 Sequence declaration using a slice

```
import java.lang.System;
def source = [0 .. 100];
var slice1 = source[0 .. 10];
var slice2 = source[0 ..< 10];
var slice3 = source[95..];
var slice4 = source[95..<];
var format = "%s = %d to %d%n";
System.out.printf(format, "slice1",
    slice1[0], slice1[(sizeof slice1)-1] );
System.out.printf(format, "slice2",
    slice2[0], slice2[(sizeof slice2)-1] );
System.out.printf(format, "slice3",
    slice3[0], slice3[(sizeof slice3)-1] );
System.out.printf(format, "slice4",
    slice4[0], slice4[(sizeof slice4)-1] );
```

```
slice1 = 0 to 10                                                    A
slice2 = 0 to 9                                                     A
slice3 = 95 to 100                                                  A
slice4 = 95 to 99                                                   A
 A  Just the start/end values
```

Here the double dot syntax creates a slice of an existing sequence, `source`. The numbers defining the slice are element indexes, so in the case of `slice1` the range `[0..10]` refers to the first eleven elements in `source`, resulting in the values 0 to 10.

The '`..`' syntax describes an inclusive range, while the '`..<`' syntax can be used to define an exclusive range (0 to 10, not including 10 itself.) If you miss the trailing delimiter off a '`..`' range, the slice will be taken to the end of the sequence–effectively making the end delimiter the sequence size minus one. If you miss the trailing delimiter off a '`..<`' range the slice will be taken to the end of the sequence minus one element–effectively dropping the last index.

2.5.4 Sequence creation using a predicate

The final weapon in the sequence arsenal (and perhaps the most powerful) is the predicate syntax, which allows us to take a conditional slice from inside another sequence. The predicate syntax takes the form of a variable and a condition separated by a bar character. The destination (output) sequence is constructed by loading each element in the source sequences into the variable, and applying the condition to determine whether it should be included in the destination or not.

Listing 2.24 Sequence declaration using a predicate

```
def source2 = [0 .. 9];
```

```
var lowVals = source2[n|n<5];
println("lowVals = {lowVals.toString()}");

def people =
    ["Alan","Andy","Bob","Colin","Dave","Eddie"];
var peopleSubset =
    people[s | s.startsWith("A")].toString();
println("predicate = {peopleSubset}");
```

```
lowVals = [ 0, 1, 2, 3, 4 ]
predicate = [ Alan, Andy ]
```

Take a look at how `lowVals` is created in listing 2.24. Each of the numbers in `source2` is assigned to `n`, and the condition `n<5` is tested to determine whether the value will be added to `lowVals`. The second example applies a test to see if the sequence element begins with the character 'A', meaning in our example only "Alan" and "Andy" will make it into the destination sequence.

Predicates are pretty useful, particularly as their syntax is nice and compact. But even this isn't the end of what we can do with sequences.

2.5.5 Sequence manipulation (insert, delete, reverse)

Sequences can be manipulated by inserting and removing elements dynamically. We can do this either to the end of the sequence, before an existing element, or after an existing element. The three variations are demonstrated with this piece of code:

Listing 2.25 Sequence manipulation: insert

```
var seq1 = [1..5];
insert 6 into seq1;
println("Insert1: {seq1.toString()}");
insert 0 before seq1[0];
println("Insert2: {seq1.toString()}");
insert 99 after seq1[2];
println("Insert3: {seq1.toString()}");
```

```
Insert1: [ 1, 2, 3, 4, 5, 6 ]
Insert2: [ 0, 1, 2, 3, 4, 5, 6 ]
Insert3: [ 0, 1, 2, 99, 3, 4, 5, 6 ]
```

This quick example (listing 2.25) shows a basic range sequence created with the values 1 through 5. The first insert appends a new value, 6, to the end of the sequence, the next inserts a new value, 0, before the current first value, and the final insert shoehorns a value, 99, after the third element in the sequence.

Listing 2.26 Sequence manipulation: delete

```
var seq2 = [[1..10],10];
delete seq2[0];
println("Delete1: {seq2.toString()}");
delete seq2[0..2];
println("Delete2: {seq2.toString()}");
delete 10 from seq2;
println("Delete3: {seq2.toString()}");
```

```
delete seq2;
println("Delete4: {seq2.toString()}");
```

```
Delete1: [ 2, 3, 4, 5, 6, 7, 8, 9, 10, 10 ]
Delete2: [ 5, 6, 7, 8, 9, 10, 10 ]
Delete3: [ 5, 6, 7, 8, 9 ]
Delete4: [ ]
```

It should be obvious what listing 2.26 does. Starting with a sequence of 1 through 10, plus another 10, the first delete operation removes the first index, the second deletes a range from index positions 0 to 2 (inclusive), the third removes any tens from the sequence, and the final delete simple removes the entire sequence.

One final trick is the ability to reverse the order of a sequence.

Listing 2.27 Sequence manipulation: reverse

```
var seq3 = [1..10];
seq3 = reverse seq3;
println("Reverse: {seq3.toString()}");
```

```
Reverse: [ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 ]
```

And finally a simple example (listing 2.27) to end the section on: the second line flips the order of the sequences, from 1 through 10, to 10 through 1.

And that's it for sequences, at least until we consider the `for` loop later on. In the next section we'll start to look at perhaps JavaFX's most powerful syntax feature, binding.

2.5.6 Sequences, behind the scenes

I hinted briefly in the introduction to sequences of how they're not really arrays. Indeed, I am indebted to Sun engineer Jasper Potts, who pointed out the folly of drawing too close an analogy between JFX sequences and Java arrays and collections.

Behind the scenes sequences use a host of different strategies to form the data you and I actually work with. Sequences are immutable (they cannot be changed, modifications result in new sequence objects) but this does not make them slow or inefficient. When we create a number range, like `[0..100]` for example, only the bounds of the range are stored, and the *n'th* value is calculated whenever it is accessed. By supporting different composite techniques, a sequence can hold a collection of different types of data represented with different strategies, and can add/remove elements within the body of the sequence without wholesale copying.

Suffice to say, when we reversed the sequence at the end of the last section (listing 2.27) no data was actually re-arranged!

More details

Michael Heinrichs has an interesting blog entry covering, in some detail, the types of representations and strategies sequences use beneath the surface to give the maximum flexibility, with minimum drudgery.

http://blogs.sun.com/michaelheinrichs/entry/internal_design_of_javafx_sequences1

2.6 Auto-updating related data, with binds

Binding is a way of defining an automatic update relationship between data in one part of your program, and data elsewhere it depends upon. Although binding has many applications, it's particularly useful in user interface programming.

Writing code to ensure a GUI always betrays the true state of the data it is representing can be a thankless task. Not only is the code which links model and UI usually verbose, quite often it lives miles away from either of them. Binding connects the interface directly to the source data, or (more accurately) to a nugget of code which interprets the data. The code which controls the interface's state is defined in the interface declaration itself!

Because binding is such a useful part of JavaFX, in the coming sections we'll cover not only its various applications but also the mechanics of how it works, for those occasions when it's important to know.

2.6.1 Binding to variables (bind)

Let's start with the basics. Here's a pretty straight forward example:

Listing 2.28 Binding into a string

```
var percentage:Integer;
def progress = bind "Progress: {percentage}% finished";
for(v in [0..100 step 20])                                          A
{   percentage = v;                                                 A
    println(progress);                                              A
}                                                                   A
```

```
Progress: 0% finished
Progress: 20% finished
Progress: 40% finished
Progress: 60% finished
Progress: 80% finished
Progress: 100% finished
```

A This is a for loop

The simple example in listing 2.28 updates the variable `percentage` from 0 to 100 in steps of 20 using a 'for' loop (which we'll study next chapter), with the variable `progress` automatically tracking the updates.

You'll note the use of the `bind` keyword, which tells the JavaFX Script compiler that the code which follows is a bound *expression*. Expressions bits of code which return values, so what `bind` is saying is "the value of this variable is controlled by this piece of code". In listing 2.28 the bound `progress` string re-evaluates its contents each time the variable it depends upon changes. So, whenever `percentage` changes, its new value is automatically reflected in the `progress` string.

But hold on, how can `progress` change when it's declared using `def`, not `var`? Well, technically it doesn't ever change. It's value changes, true, but its *real* contents (the expression) never actually get reassigned. This is why, back when we covered `var` and `def`,

I warned against thinking of `def` variables as the simple constants some other languages have. As a one-way bound variable should not be *directly* assigned to, using `def` is more correct than using `var`.

Bind doesn't just work with strings, but number types too, as we'll see next.

Listing 2.29 Binding between variables

```
var thisYear = 2008;
def lastYear = bind thisYear-1;
def nextYear = bind thisYear+1;
println("2008: {lastYear}, {thisYear}, {nextYear}");
thisYear = 1996;
println("1996: {lastYear}, {thisYear}, {nextYear}");
```

```
2008: 2007, 2008, 2009
1996: 1995, 1996, 1997
```

In listing 2.29 the values of `lastYear` and `nextYear` are dependent on the current contents of `thisYear`. A change to `thisYear` causes its siblings to re-calculate the expression associated with their binding, meaning they will always be one year behind and ahead whatever `thisYear` is set to.

2.6.2 Binding to bound variables

What about bound variables themselves, can they be the target of other bound variables, creating a chain of bindings? The answer, it seems, is yes!

Listing 2.30 Binding to bound variables

```
var flagA = true;
def flagB = bind not flagA;
def flagC = bind not flagB;
println("flagA = {flagA}, "
    "flagB = {flagB}, flagC = {flagC}");
flagA = false;
println("flagA = {flagA}, "
    "flagB = {flagB}, flagC = {flagC}");
```

```
flagA = true, flagB = false, flagC = true
flagA = false, flagB = true, flagC = false
```

The first two variables in listing 2.30, `flagA` and `flagB`, will always hold opposite values. Whenever `flagA` is set, its companion is set to the inverse automatically. The third value, `flagC`, is the inverse of `flagB`, creating a chain of updates from A to B to C, such that C is always the opposite of B and the same as A.

2.6.3 Binding to a sequence element

Perhaps you're wondering how to use the bind syntax with a sequence? As luck would have it, that's the next bit of example code:

Listing 2.31 Binding against a sequence element

```
var range = [1..5];                                                    A
```

```
def reference = bind range[2];
println("range[2] = {reference}");
delete range[0];                                                    B
println("range[2] = {reference}");
delete range;                                                       C
println("range[2] = {reference}");
```

```
range[2] = 3
range[2] = 4
range[2] = 0
 A  'range' is [1,2,3,4,5]
 B  'range' is [2,3,4,5]
 C  'range' is empty
```

When we bind against a sequence element we do so by way of its index–when the sequence is extended or truncated the bind does not track the change by adjusting its index to match. In the above example (listing 2.31), even though the first element of `range` is deleted, causing the other elements to *move down* the sequence, the bind still points to the third index. Also, as we'd expect, accessing the third index after all elements have been deleted returns a default value.

2.6.4 Binding to an entire sequence (for)

In the previous section we witnessed what happens when we bind against an individual sequence element. But what happens when we bind against an entire sequence?

Listing 2.32 Binding to a sequence itself

```
var multiplier:Integer = 2;
var seqSrc = [ 1..3 ];
def seqDst = bind for(x in seqSrc) { x*multiplier; }
println("seqSrc = {seqSrc.toString()},"
     " seqDst = {seqDst.toString()}");
insert 10 into seqSrc;                                              A
println("seqSrc = {seqSrc.toString()},"
     " seqDst = {seqDst.toString()}");
multiplier = 3;                                                     B
println("seqSrc = {seqSrc.toString()},"
     " seqDst = {seqDst.toString()}");
```

```
seqSrc = [ 1, 2, 3 ], seqDst = [ 2, 4, 6 ]
seqSrc = [ 1, 2, 3, 10 ], seqDst = [ 2, 4, 6, 20 ]
seqSrc = [ 1, 2, 3, 10 ], seqDst = [ 3, 6, 9, 30 ]
 A  Change source sequence
 B  Change multiplier
```

Listing 2.32 shows one sequence, `seqDst`, bound against another, `seqSrc`. The bind expression takes the form of a loop which doubles the value in each element of the source sequence. When the source sequence changes, for example a new element is added, the bind ensures the destination sequence is kept in step, using the expression.

When the `multiplier` changes the destination sequence is again kept in step. So, both `seqSrc` and `multiplier` affect `seqDst`. Binds are sensitive to change from every

variable within their expression, something to keep in mind when writing your own bind code.

2.6.5 Binding to code

In truth all the examples thus far have demonstrated binding to code–a simple variable read is, after all, code. The bind keyword merely attaches an *expression* (any unit of code which produces a result) to a read-only variable. This section looks a little deeper into how to exploit this functionality, for example: what if we need some logic to control how a bound variable gets updated?

Listing 2.33 Binding to a ternary expression

```
var mode = false;
def modeStatus = bind if(mode) "On" else "Off";
println("Mode: {modeStatus}");
mode = true;
println("Mode: {modeStatus}");
```

```
Mode: Off
Mode: On
```

Listing 2.33 contains a binding which uses an if/else block to control the update of the string modeStatus, resulting in either the contents “on” or “off”. The if/else block is re-evaluated each time mode changes.

Next we'll see an even more ambitious example.

Listing 2.34 A more complex binding example

```
var userID = "";
def realName = bind getName(userID);
def status = bind getStatus(userID);
def display = bind "Status: {realName} is {status}";
println(display);
userID = "adam";
println(display);
userID = "dan";
println(display);

function getName(id:String) : String                                    A
{   if(id.equals("adam")) { return "Adam Booth"; }                      A
    else if(id.equals("dan")) { return "Dan Jones"; }                   A
    else { return "Unknown"; }                                          A
}                                                                       A
                                                                        A
function getStatus(id:String) : String                                  A
{   if(id.equals("adam")) { return "Online"; }                          A
    else if(id.equals("dan")) { return "Offline"; }                     A
    else { return "Unknown"; }                                          A
}                                                                       A
```

```
Status: Unknown is Unknown
Status: Adam Booth is Online
Status: Dan Jones is Offline
```

A Functions, accept and return a String

We haven't covered functions yet, but it shouldn't be hard to figure out what the code in listing 2.34 is doing. Working forward, down the chain, we start with `userID`, bound by two further variables called `realName` and `status`. These two variables call functions with `userID` as a parameter, both of which return a `String` whose contents depend upon the `userID` passed in. Finally a string called `display` adds an extra layer of binding by using the two bound variables to form a status string. Merely changing `userID` causes `realName`, `status` and `display` to automatically update.

2.6.6 Bidirectional binds (with inverse)

We've seen some pretty complex binding examples thus far, but suppose we just wanted a simple bind which directly mirrored another variable, would it be possible to have the bind work in both directions, so a change to either variable would result in a change to its *twin*?

Listing 2.35 Bidirectional binding

```
var dictionary = "US";
var thesaurus = bind dictionary with inverse;
println("Dict:{dictionary} Thes:{thesaurus}");

thesaurus = "UK";
println("Dict:{dictionary} Thes:{thesaurus}");

dictionary = "FR";
println("Dict:{dictionary} Thes:{thesaurus}");
Dict:US Thes:US
Dict:UK Thes:UK
Dict:FR Thes:FR
```

In listing 2.35 we use the `with inverse` keywords to create a two way link between the variables `dictionary` and `thesaurus`. A change to one is automatically pushed across to the other. We can only do this because the bound expression is a simple, one to one, reflection of another variable.

This may not seem like the most exciting functionality in the world, admittedly the short listing above doesn't really do the idea justice. Imagine a data structure, who's members are displayed on screen in editable text fields. When the data structure changes the text fields should change. When the text fields change the data structure should change. You can imagine the fun we'd have implementing that with a unidirectional bind: whichever end we put the bind on would become beholden to the bind expression, incapable of being altered. Bidirectional binds neatly solve that problem–both ends are editable.

By the way, did you spot anything unusual about the `thesaurus` variable? (Well done if you did!) Yes, we're using `var` instead of `def`. Unidirectional binds could not, by their very nature, be changed once declared. But bidirectional binds, by their very nature, are intended to allow change. Thus `var`, instead of `def`.

2.6.6 The mechanics behind bindings

The OpenJFX project's own language documentation goes into some detail on how binding works, and possible side effects which may occur if a particular set of circumstances conspire.

The first thing you should note is that you aren't free to put just any old code inside a bind's body–it has to be an expression (a single unit of code which returns a value.) This means, for example, you can't assign or update the value of another variable directly within the block of code associated with a bind. Makes sense–the idea is to define relationships between data sources and their interfaces; bindings are not general purpose triggers for running code when variables are accessed.

> **Expression mystery**
>
> Eagle eyed readers may be saying at this point "hold on, in previous sections we saw binds working against conditions and loops, yet they aren't in themselves expressions!" Well actually, you'd be wrong if you thought that–in JavaFX Script *they are* expressions! But a full exploration of this will have to wait until next chapter.

The only exception to the expression rule is variable declarations. You are allowed, it seems, to declare new variable definitions within a bound block. However you cannot update the variable once declared, so use the `def` keyword. (The SDK 1.0 compiler allows both `var` and `def` keywords to be used, but the language documentation appears to suggest only `def` will be supported in future.)

Moving on to our second topic: this term "recalculation", what does it *really* mean? When it comes to bound expressions JavaFX Script attempts to perform a *minimal recalculation*, which sounds rather cryptic, but only means it's as lazy as possible. An example will explain all.

Listing 2.36 Minimal recalculation

```
var x = 0;
def y = bind getVal() + x;
for(loop in [1..5])
{   x = loop;
    println("x = {x}, y = {y}");
}

function getVal() : Integer
{   println("getVal() called");
    return 100;
}
```

```
getVal() called
x = 1, y = 101
x = 2, y = 102
x = 3, y = 103
x = 4, y = 104
x = 5, y = 105
```

The bind in listing 2.36 has two parts, a call to a function, `getVal()`, and a variable, `x`. As `x` gets updated in the for loop you might expect the `println()` call in `getVal()` to be triggered repeatedly–but it's only called once. The result from the first half of the expression (the function call) was reused, as JavaFX knows it's not dependent on the value which caused the update. This speeds things along, keeping the execution tight, but it could have unexpected side effects if your code assumes every part of a bound expression will always run. Best practice, therefore, dictates avoiding such assumptions.

2.6.7 Bound functions (bound)

In the previous section we saw how bindings try to perform the minimal amount of recalculation necessary for each update. This is fine when the expression we're binding against is transparent–but what if the bind is against some *black box* piece of code, with outside dependencies?

Functions provide such a black box. When a bind involves a function it only 'watches' the parameters passed into the function to determine when recalculation is needed. The mechanism inside the function (the code it contains) is not considered, and as such any dependencies it may have are not factored into the bind recalculations. An example shows both the problem and the solution:

Listing 2.37 Bound functions

```
var ratio = 5;
var posX = 5;
var posY = 10;
def coords1 = bind "{scale1(posX)},{scale1(posY)}";
def coords2 = bind "{scale2(posX)},{scale2(posY)}";

function scale1(v:Integer) : Integer
{   return v*ratio;
}
bound function scale2(v:Integer) : Integer
{   return v*ratio;
}

println("1={coords1}  2={coords2}");
posX = 6;
println("1={coords1}  2={coords2}");
posY = 9;
println("1={coords1}  2={coords2}");
ratio = 3;
println("1={coords1}  2={coords2}");
```

```
1=25,50  2=25,50
1=30,50  2=30,50
1=30,45  2=30,45
1=30,45  2=18,27
```

Listing 2.37 shows two binds, `coords1` and `coords2`, which both call a function. We'll look at `coords1` first, as this is the one presenting the problem.

The variable `coords1` is updated via two calls to a function called `scale1()`. The function is used to scale two coordinates, `posX` and `posY`, by a given factor, `ratio`. The `scale1()` function accepts each coordinate, and scales it by the appropriate factor using a reference to the external `ratio` variable. Watch what happens as we change the three variables involved in the bind: `posX`, `posY` and `ratio`.

We change `posX`, and the value of `coords1` is automatically updated. This is because the bind knows that `posX` is integral to the bind. Likewise for `posY`. But a change to `ratio` does not force a recalculation of `coord1`. Why? The fact that this external variable is key to the integrity of the `coords1` bind is lost–the function body is a black box, and was not scanned for dependencies.

And now the solution: `coords2` uses *exactly* the same code, except the function it binds against, named `scale2()`, carries the `bound` keyword prefix. This keyword is a warning to the compiler, flagging up potential external dependencies. When JavaFX Script binds against a function marked `bound`, it looks inside for variables to include in the binding. Therefore `coord2` gets correctly recalculated when `ratio` is changed, even though the reference to `ratio` is hidden inside a function.

We call functions like `scale1()` *unbound functions*–their body is never included in the list of dependencies for a bind. We call functions like `scale2()` *bound functions*–their body is scanned for external variables they rely upon, and these are included as triggers to cause a recalculation of the bind.

2.6.8 Bound object literals

Now we've explored how binding works, and the difference between bound and unbound functions, we can look at how they respond to object literals.

Object what?!? Sorry, but this is one of those occasions when I have to skip ahead a little and introduce something we won't cover fully until next chapter. Object literals are JavaFX Script's way of declaratively creating complex objects. You should still be able to follow the basics of what follows, although you may want to bookmark this section and revisit it once you've read chapter three.

Listing 2.38 Binding and object literals

```
class TextContainer                                                    A
{   var label:String;
    var content:String;
    init
    {   println("Object created: {label}");
    }
}

var someContent:String = "Pew, Pew, Barney, Magreu";                   B

def obj1 = bind TextContainer                                          C
{   label: "obj1";
    content: someContent;
};
def obj2 = bind TextContainer                                          D
```

```
{   label: "obj2";
    content: bind someContent;
};

someContent = "Cuthbert, Dibble and Grub";                                  E
```

```
Object created: obj1
Object created: obj2
Object created: obj1
```

A Create a class
B Some text to bind to
C Causes output line 1
D Causes output line 2
E Causes output line 3

Take a look at listing 2.38 above. We start by creating a new class, with two variables. The first, `label`, merely tags each object so we can tell them apart in the output. The other, `content`, is the variable we're interested in. You'll note there's also a block of `init` code in our class–this will run whenever an object is created from the class, printing out the `label` variable so we can see when objects are created.

We then define a `String` called `someContent`, to use in our object literals. The next two blocks of code are said object literals. We create two of them, named `obj1` and `obj2`, applying a bind to both. We use the variable `someContent` to populate the `content` variable of the objects (although there's is a subtle difference between the two declarations which will alter the way binding works.)

Creating the two objects causes the `init` code to run, causing the first two lines of our output–no great surprises there. But look what happens when we change the `someContent` variable which was used by the objects. The first object (`obj1`) is recreated, while the second (`obj2`) is not. Why is that?

Think back to the way binds worked with functions. When one of the function parameters changes, the bind reruns the function with the new data. Object literals work the same way, the bind applies not only to the object, but to the variables used when declaring it. When one of those variables changes, the object literal is 'rerun' with the new data, causing a new object to be created. So that explains why a new `obj1` was created when `someContent` was changed, but why didn't `obj2` suffer the same fate?

If you look at where `someContent` is used in the declaration of `obj2`, you'll see a second `bind`. This inner bind effectively shields the outer bind from changes affecting the object's `content` variable, creating a kind of nested context which prevents the recalculation from bubbling up to the object level. Remember this trick whenever you need to bind against an object literal, but don't want the object creation 'rerun' whenever one of the variables the declaration relied upon changes.

Whew... so that was JavaFX bindings! If you don't yet appreciate how useful they are, you'll get plenty of examples in our first project in chapter four. Meanwhile, in the next section we'll finish this chapter off with perhaps JavaFX's most peculiar looking bit of syntax.

2.7 Avoiding naming conflicts, with quoted identifiers

Quoted identifiers are the bizarre double angle brackets wrapping some terms in JFX source code, for example `<<java.lang.System>>`. Strange as these symbols look, they actually perform a very simple, yet vital, function. JavaFX sits atop the Java platform, and has access to its libraries. Sometimes names from Java clash with keywords or syntax patterns in JavaFX Script. Those peculiar angle brackets resolving this discord, ensuring a blissful harmony of Java and JavaFX at all times.

The double angle brackets wrap identifiers (identifiers are the names referring to variables, functions, classes, etc.) to exclude them from presumptions the compiler might otherwise make. Listing 2.39 shows an example.

Listing 2.39 Quoted identifiers

```
var <<var>> = "A string variable called var";
println("{<<var>>}");
```

```
A string variable called var
```

Please (please please!) don't do something as stupid as the above in real production code. By using (abusing?) quoted identifiers we've created a variable with the name of a JavaFX Script keyword. Normally we'd only do this if “var” was something in Java we needed to reference.

2.8 Summary

As you've now seen, JavaFX value types and sequences house some pretty powerful features. Chief amongst them is the ability to bind data into automatically updating chains, slashing the amount of code we have to write to keep our user interface up to date. String formatting and sequence manipulation also offer great potential when used in a graphics and animation rich environment.

And talking of animation, there was one small piece of the JavaFX Script's data syntax we missed out in this chapter. JFX provides a literal syntax for quick and easy creation of points on an animation time line, using the keywords `at` and `tween`. This is quite a specialist part of the language, without application beyond of the remit of animation. Because it's impossible to demonstrate how this syntax works without reference to the graphics classes and packages it supports, I've held discussion of this one small part of the language over for a later chapter.

As interesting as this chapter was (right?), I've no doubt you're eager to see some actually code in action, like loops and classes and the like. In the next chapter we'll complete our tour of the JavaFX Script language by looking at the meat of the language, the stuff which actually gets our data moving.

3

JavaFX Script: code and structure

The previous chapter looked at JavaFX Script's data types and manipulations; this chapter looks at its code constructs. Separating the two is somewhat arbitrary—we saw plenty of code hiding in the previous chapter's examples because code and data are flip sides of the same coin. Ahead we'll see how the concepts we discovered last chapter integrate into the syntax as a whole. Our grounding in 'data' will hopefully engendering a more immediate understanding as we encounter conditions, loops, functions, classes and the like.

As mentioned briefly in chapters one and two, JavaFX Script is what's referred to as an *expression language*, meaning most executable bits of code return either zero, one or more values (aka: void, a value, or a sequence.) Even loops and conditions will work on the right hand side of an assignment. It's important to fix this idea in your mind as we work through the content of this chapter, particularly if you're not used to languages working this way. I'll point out the ramifications as the chapter unfolds.

We start with higher level language features like classes and objects, and work our way down into the trenches to examine loops and conditions. It goes without saying you'll require an understanding of the material in the previous chapter—so if you skipped ahead to the *juicy code stuff*, please consider backtracking to at least read up on *binds* and *sequences*.

I've maintained a couple of conventions from the last chapter. Firstly, source code is presented in small chunks which (unless otherwise stated) compile and run independently, with the console output presented in bold. Secondly, I'm continuing to incur the wrath of pedants by referring to the language variously as “JavaFX Script” (its proper name), “JavaFX” and “JFX”, as the mood takes me.

We have quite an exciting journey ahead, with a few exciting twists and turns, so without further ado let's get going. We'll start by looking at the highest level of structure we can impose on our code: the package.

3.1 Imposing order and control, with packages (package, import)

The outermost construct for imposing order on our code is the package. Packages allow us to relate portions of our code together for convenience, to control access to variable and functions (see *access modifiers*, later), and to permit classes to have identical names but different purposes.

Listing 3.1 Using the package statement to shorten class names

```
import java.util.Date;                                                    A
                                                                          A
var date1:Date = Date {};                                                 A
var date2:java.util.Date = java.util.Date {};                             A
```

A "Date" lives in the package "java.util"

In listing 3.1 we see two different ways of creating a `Date` object above—the first makes use of the `import` statement at the start of the code (and would fail to compile without it), while the second does not. As in Java, an asterisk can be used at the end of an `import` statement instead of a class name, to include all the classes from the stated package without having to list them individually. But how do we create our own packages?

Listing 3.2 Including a class inside a package

```
package jfxia.chapter3;                                                   A
                                                                          A
public class Date                                                         A
{   override function toString() : String                                 A
    {   "This is our date class";                                         A
    }                                                                     A
};                                                                        A
```

A Should go in the file "Date.fx"

As demonstrated in listing 3.2, the `package` statement, which must appear at the start of the source, places the code from this example into a package called `jfxia.chapter3`, from where `import` may be used to pull it into other class files.

For non-Java programmers there's a more in depth discussion of packages in Appendix C, section C3. Next we turn our attention to the class content itself.

3.2 Developing classes

Classes are an integral part of Object Orientation, encapsulating state and behavior for components in a larger system, allowing us to express software through the relationships linking its autonomous component parts. Object Orientation has become an incredibly popular way of constructing software in recent years: both Java, and its underlying Java Virtual Machine environment, are heavily object centric. No surprise, then, that JavaFX Script is also Object Oriented!

JavaFX's implementation of OO is close, but not identical to, Java's—for example, JavaFX permits multiple inheritance In the coming sub-sections we'll explore the ins and outs of JFX classes, and how to define, create, inherit, control and manipulate them.

3.2.1 Scripts

In some languages source code files are just arbitrary containers for code. In other languages the compiler attaches significance to where the code is placed, as with the Java compiler's linking of class with source files. In JavaFX Script, however, the source file actually has a role in the language itself.

In JFX a single source file is known as a *script*, and scripts can have their own code, functions and variables, distinct from a class. They can also be used to create an application's top level code – the stuff that runs when your program starts.

Listing 3.3 Scripts and classes

```
// This code lives in a file called "Examples2.fx"
package jfxia.chapter3;

def scriptVar:Integer = 99;                                          A

function scriptFunc() : Integer                                      B
{   def localVar:Integer = -1;                                       B
    return localVar;                                                 B
}                                                                    B

println                                                              C
(   "Examples2.scriptVar = {Examples2.scriptVar}\n"                  C
    "Examples2.scriptFunc() = {Examples2.scriptFunc()}\n"            C
    "scriptVar = {scriptVar}\n"                                      C
    "scriptFunc() = {scriptFunc()}\n"                                C
);                                                                   C
```

```
Examples2.scriptVar = 99
Examples2.scriptFunc() = -1
scriptVar = 99
scriptFunc() = -1
```

A Script variable
B Script function
C Bootstrap code

Listing 3.3 shows variables and functions in the script context. We saw plenty of examples of this type of code in chapter two, where almost every listing used the script context for its code.

Script functions and variables live outside of any class. In many ways they behave like static variables in Java. They can be accessed by using the script name as a prefix, like `Examples2.scriptVar` or `Examples2.scriptFunc()`, although when accessed from inside their own script (as listing 3.3 shows) the prefix can be omitted. They are visible to all code inside the current script, including classes. External visibility (outside the script) is controlled by *access modifiers*, which we'll study later in this chapter.

When the JavaFX compiler runs it turns each script into a bytecode class (plus an interface, but we won't worry about implementation details here!) The script context functions and variables effectively become what Java would term static methods and variables in said class – but the script context can also contain loose code which isn't inside a function. What happens to this code?

It gets bundled up and executed if we run the script. In effect, it becomes Java's `public static main()` method. In listing 3.3 the `println()` will run if we launch the class `jfxia.chapter3.Examples2` using the JavaFX runtime. There is, however, one restriction on loose expressions: they can only be used in scripts with no externally visible variables, functions or classes. This is another *access modifiers* issue which will be explained in a few pages time.

Having seen scripts in action, it's about time we looked at some class examples.

3.2.2 Class definition (class, def, var, function, this)

Creating new classes is an important part of Object Orientation, and true to form the JavaFX Script syntax boasts its trademark brevity. In this section we'll forgo mention of inheritance for now, concentrating instead on the basic format of a class, its data and behavior.

We've seen script context variables and functions in the last section (and last chapter too), and their class equivalents are no different. The official JFX language documentation refers to them as *instance variables* and *instance functions*, because the are accessed via a class instance (an object of the class type), so we'll stick to the same naming convention. Also, to make the prose flow more readily, I'll sometimes refer to functions and variables using the combined term *members*, as in “script members”, or “instance members”.

Listing 3.4 Class defintion, with variables and functions

```
class Track                                                          A
{   var title:String;                                                B
    var artist:String;                                               B
    var time:Duration;                                               B

    function equals(t:Track) : Boolean                               C
    {   return (t.title.equals(this.title) and                       C
            t.artist.equals(this.artist));                           C
    }                                                                C

    override function toString() : String                            D
    {   return '"{title}" by "{artist}" : '                          D
            '{time.toMinutes() as Integer}m '                        D
            '{time.toSeconds() mod 60 as Integer}s';                 D
    }                                                                D
}

var song:Track = Track                                               E
{   title: "Special"                                                 E
    artist: "Garbage"                                                E
    time: 220s                                                       E
};                                                                   E
println(song);
```

```
"Special" by "Garbage" : 3m 40s
```

A Track class
B Instance variables
C Instance function
D Another instance function

E Declaring a new object from Track

The class above, in listing 3.4, defines a music track with three variables and two functions. You'll note JavaFX Script uses the same naming conventions as Java: class names begin with a capital letter, function and variable names do not. Both use camel case to form names from multiple words. You don't have to stick to these rules, but it helps make your code readable to other programmers.

The `title`, `artist` and `time` are the variables which objects of type `Track` will have, the instance variables. Just like script variables they can be assigned initial values, although the three examples in listing 3.4 all use defaults.

Properties: what's in a name?

Although the JavaFX Script documentation prefers the formal term "instance variables", sometimes class level variables are referred to as "properties". In programming a property is a class element who's read/write syntax superficially resembles a variable, but with code which runs on each access. The result is much cleaner code than, for example, Java's verbose getter/setter method calls (although the function is the same).

JavaFX Script variables can be bound to functions controlling their value, and triggers may run when their value changes. Yet binds and triggers are not intended as a property syntax. The real reason JavaFX Script's variables are sometimes called properties likely has more to do with JFX's declarative syntax giving the same clean code feel as the *real* properties found in languages like C#.

Functions in JFX classes are defined using the keyword `function`, followed by the function's name, a list of parameters and their type in parenthesis, a colon and the return type (where `Void` means no return.) There are two functions above, the first accepts another `Track` and checks if it references the same song as the current object, the second constructs a `String` to represent this song. The `toString()` function has the keyword `override` prefixing its definition. This is a requirement of JavaFX Script's inheritance mechanism, which we'll look at later this chapter.

The keyword `this` can be used inside the class to refer to the current object, although it's use can usually be inferred, as the `toString()` function demonstrates. A class's instance members are accessible to all code inside the class, and the enclosing script too via objects of the class. External access (outside the script) is controlled by access modifiers (see later).

Once we've defined our class, we can create objects from it. We'll look at different ways of doing that in the next section, but just to complete our example I've included a sneak preview at the tail of listing 3.4. In the code, `song` is created as an object of type `Track`. In case you're wondering, we didn't strictly need to specify the type after `song` (*duck typing* would work) but as this is your first proper class example I thought it wouldn't hurt to go that extra mile.

Before we move on, there's some bits and pieces which need extra attention. Take a look at the source code below:

Listing 3.5 A closer look at functions

```
// More on functions
function doesReturn() : String
{   "Return this";
}
function doesNotReturn()
{   var discarded = doesReturn();
}
```

A couple of things to note in listing 3.5, above. Firstly the `doesNotReturn()` function fails to declare its return type. It seems explicitly declaring the return type is unnecessary when the function doesn't return anything—`Void` is assumed.

Secondly, and far more importantly, shouldn't `doesReturn()` have a `return` keyword in there somewhere? Recall, JavaFX Script is an expression language, a simple expression on its own is sufficient to return a value. The last expression of a function can serve as its return value.

3.2.3 Object declaration (init, postinit, isInitialized(), new)

We saw an example of creating an object from a class in the previous section. Many other Object Oriented languages call a constructor, often via a keyword such as `new`, but JavaFX Script objects have no constructors, preferring the trademark declarative syntax.

By invoking what the JFX documentation snappily calls the *object literal* syntax, we can declaratively create new objects. Simply state the class name, open some curly braces, list each instance variable we want to give an initial value to (using its name and value separated by a colon), and don't forget the closing 'curly'.

Listing 3.6 Object declaration, using declarative syntax or the new keyword

```
// Source file: SpaceShip.fx
package jfxia.chapter3;

def UNKNOWN_DRIVE:Integer = 0;                                          A
def WARP_DRIVE:Integer = 1;                                             A

class SpaceShip                                                         B
{   var name:String;                                                    B
    var crew:Integer;                                                   B
    var canTimeTravel:Boolean;                                          B
    var drive:Integer = SpaceShip.UNKNOWN_DRIVE;                        B
                                                                        B
    init                                                                B
    {   println("Building: {name}");                                    B
        if(not isInitialized(crew))                                     B
            println("  Warning: no crew!");                            B
    }                                                                   B
    postinit                                                            B
    {   if(drive==WARP_DRIVE)                                           B
            println("  Engaging warp drive");                          B
```

```
    }                                                                    B
}                                                                        B

def ship1 = SpaceShip                                                    C
{   name:"Starship Enterprise"                                           C
    crew:400                                                             C
    drive:SpaceShip.WARP_DRIVE                                           C
    canTimeTravel:false                                                  C
};                                                                       C

def ship2 = SpaceShip                                                    D
{   name:"The TARDIS" ; crew:1 ; canTimeTravel:true                      D
};                                                                       D

def ship3 = SpaceShip{ name:"Thunderbird 5" };                           E

def ship4 = new SpaceShip();                                             F
ship4.name="The Liberator";                                              F
ship4.crew=7;                                                            F
ship4.canTimeTravel=false;                                               F

Building: Starship Enterprise
  Engaging warp drive
Building: The TARDIS
Building: Thunderbird 5
  Warning: no crew!
Building:
  Warning: no crew!
```

A Script variables
B Our class, including init / postinit blocks
C Declarative syntax
D Again, declarative syntax
E Only name set
F Java-style syntax

In listing 3.6 we create four objects: the first three using the JavaFX Script syntax, and the final one using a Java-style syntax.

From the first three examples you'll note how JFX uses the colon notation, allowing us to set any available variable on the object as part of its creation. This way of explicitly writing out objects is what's referred to as an *object literal*. The second example uses a more compact layout: multiple assignments per line, separated by semi-colons.

Before looking at the third example, we need to consider the `init` and `postinit` blocks inside the class. As you may have guessed, these run when an object is created. The sequence of events is:

- The virgin object is created.
- The Java superclass default constructor is called.
- The object literal's instance variable values are computed, **but not set**.
- The instance variables of the JavaFX superclass are set.
- The instance variables of this class are now set, in lexical order.
- The `init` block is called, if present. The object is now considered initialized.

- The `postinit` block is called, if present.

The `isInitialized()` built-in function allows us to test whether a given variable has been explicitly set. In our third example only the name variable is set in the object literal, so the warning message tells us that Thunderbird 5 has no crew (which, in itself, might demand the attention of International Rescue!) Conveniently, `isInitialized()` isn't fooled by the fact that `crew`, as a value type, will have a default (unassigned) value of zero.

The `isInitalized()` function is handy for knowing whether an object literal bothered to set an instance variable, so we can assign appropriate initial values to those variables it missed. Alternatively, you may provide multiple means of configuring an object, like separate `lengthCm` and `lengthInches` variables, and want to know which was used.

Moving on to the fourth example, you'll note it looks like the way we create new object instances in Java. Indeed, that's intentional. There may be times when we are forced to instantiate a Java object using a specific constructor, the `new` syntax allows us to do just that. But `new` can be used on any class, including JavaFX Script classes, although we should resist that temptation. The example above is *bad practice*! The `new` syntax should only be used when JavaFX Script's declarative syntax will not work.

Because the fourth object's instance variables don't get set until after the object is created, the "Building" message has an empty `name` and the `crew` warning is triggered.

3.2.4 Object declaration and sequences

There's actually nothing special about the example we're about to see. It simply pulls together the object creation syntax we just saw above, with the sequence syntax we witnessed last chapter. It's worth an example on its own, however, as this mixing of objects and sequences crops up frequently in JavaFX programming.

The code below creates two `Track` objects inside a `Track[]` sequence. Note: to run this code you need the `Track` class we saw a short while back.

Listing 3.7 Sequence declaration

```
var playlist:Track[] =
[   Track
    {   title: "Special"
        artist: "Garbage"
        time: 220s
    } ,
    Track
    {   title: "End of the World..."
        artist: "REM"
        time: 245s
    }
];
println(playlist.toString());

[ "Special" by "Garbage" : 3m 40s, "End of the World..."
[CA]by "REM" : 4m 5s ]
```

The above cyan [CA]'s are code continuations, for wrapped lines.

Instead of a boring list of comma separated numbers or strings, in listing 3.7 the sequence contains object declarations between its square brackets. For readability I've used only two tracks in `playlist`, but the sequence could hold as many as you want—although be careful, the JavaFX compiler may issue warnings if you attempt to add anything by Rick Astley.

3.2.5 Class inheritance (abstract, extends, override)

One of the most important tenets of Object Orientation is *subclassing*, the ability of a class to inherit fields and behavior from another. By defining classes in terms of one another we make our objects amenable to *polymorphism*—allowing them to be treated as more than one type.

For readers unfamiliar with terms like "polymorphism", there's a brief beginners guide to Object Orientation in Appendix C, section C5.

Given its origins, you may be surprised to learn JavaFX supports multiple inheritance: the ability to inherit from more than one class at a time. Multiple inheritance comes with one major caveat, however, which we'll study in depth next section. In this section we'll get familiar with the basics. The following example has been broken up into parts, which we'll explore piece by piece.

Listing 3.8 Class inheritance, part 1

```
import java.util.Date;

abstract class Animal
{   var life:Integer = 0;                                          A
    var birthDate:Date;                                            A

    function born() : Void                                         B
    {   this.birthDate = Date{};                                   B
    }                                                              B
                                                                   B
    function getName() : String                                    B
    {   "Animal"                                                   B
    }                                                              B

    override function toString() : String                          C
    {   "{this.getName()}  Life: {life}  "                         C
            "Bday: {%te birthDate} {%tb birthDate}";               C
    }                                                              C
}
```

A Instance variables
B Instance functions, new for this class
C Instance function, inherited

Listing 3.8 shows a simple `Animal` class. It's home to just a life gage and a date of birth, plus three functions: `born()` for when the object is just created, `getName()` to get the animal type as a `String`, and `toString()` for getting a printable description of the object. This is the base class onto which we'll build in the following parts of the code.

The `abstract` keyword prefixing the class head tells the compiler objects of this class cannot be created directly. Sometimes we need a class to only be a base (parent) class to other classes. We can't create `Animal` objects directly, but we can subclass `Animal` and create objects of the subclass. Thanks to polymorphism we can even assign these subclass objects to a variable of type `Animal`, although handling objects as type `Animal` is not the same as creating an `Animal` object itself. We'll see an example of this shortly.

Listing 3.9 Class inheritance, part 2

```
class Mammal extends Animal                                          A
{   override function getName() : String                             B
    {   "Mammal"                                                     B
    }                                                                B

    function giveBirth() : Mammal                                    C
    {   var m  = Mammal { life:100 };                                C
        m.born();                                                    C
        return m;                                                    C
    }                                                                C
}
```

A Subclass of Animal
B Overrides the one in Animal
C Brand new instance method

Our second chunk of code (listing 3.9, above) inherits the first, by using the `extends` keyword after its name followed by the parent class, in this case `Animal`. As you may expect, this makes `Mammal` a type of `Animal`.

The class overrides the `getName()` method to provide its own answer, which explains the `override` keyword prefixing the function. The extra keyword doesn't do anything, other than help document the function and make it harder for bugs to creep into your code. If you miss it off you'll get a compiler warning. If you include it when you shouldn't, you'll get a compiler error.

You should use the `override` keyword even when subclassing Java classes, which is why you may have spotted it on the `toString()` function of `Animal`, in listing 3.8. All objects in the Java Virtual Machine are descendants of `java.lang.Object`, which means all JavaFX objects are too, even if they don't explicitly extend any class. Thus `toString()`, which originates in `Object`, needs the `override` keyword.

The `Animal` class adds an extra function for giving birth to a new `Mammal`. The new function creates a fresh `Mammal`, sets its initial `life` value, then calls `born()`. The `born()` function is inherited from `Animal`, along with the `toString()` function.

So far so good, how another another `Animal` subclass?

Listing 3.10 Class inheritance, part 3

```
class Reptile extends Animal
{   override var life =  200;                                        A

    override function getName() : String                             B
    {   "Reptile"                                                    B
```

```
    }                                                                    B

    function layEgg() : Egg                                              C
    {   var e = Egg                                                      C
        {   baby: Reptile {}                                             C
        };                                                               C
        e;                                                               C
    }                                                                    C
}

class Egg                                                                D
{   var baby:Reptile;                                                    D
    function toString() : String                                         D
    {    return "Egg => Baby:{baby}";                                    D
    }                                                                    D
}                                                                        D
```

A Override inherited initial value
B Overrides the one in Animal
C Create a Reptile inside an Egg
D The Egg class itself

Again we have a subclass of Animal, this time called `Reptile` (see listing 3.10), with its own overridden implementation of `getName()` and its own new function. The new function in question creates and returns a fourth type of object, `Egg`, housing a new `Reptile`.

At the head of the `Reptile` class is an overridden instance variable. Why do we need to override an instance variable? Think about it: overriding an instance function redefines it, replacing its (code) contents. Likewise, overriding an instance variable redefines its contents, giving it a new initial value. This means any `Reptile` object literal failing to set `life` will get a value of 200, not 0 (the value inherited from `Animal`.)

Before we move on, check out the use of JFX's declarative syntax in `layEgg()`. The `Reptile` object is literally constructed inside the `Egg`. We could have done it long hand (the Java way), creating a `Reptile`, then the `Egg`, then plugging one into the other. But the JavaFX Script syntax allows us far more elegance.

Listing 3.11 Virtual functions demonstrated

```
def mammal = Mammal { life:150  ; birthDate: Date{} };
def animal:Animal = mammal.giveBirth();
println(mammal);                                                         A
println(animal);                                                         B
println(animal.getName());                                               C

def reptile = Reptile { life:175  ; birthDate: Date{} };
def egg = reptile.layEgg();
println(reptile);                                                        D
println(egg);                                                            E

Mammal  Life: 150  Bday: 3 Aug
Mammal  Life: 100  Bday: 3 Aug
Mammal
Reptile  Life: 175  Bday: 3 Aug
Egg => Baby:Reptile  Life: 200  Bday: null null
```

A First output line

B Second output line
C Third output line
D Fourth output line
E Final output line

Now finally some code to test our new objects. In listing 3.11 we create two `Mammal` objects, but here's the clever part: we store one of them as an `Animal`. Even though `Animal` is abstract, and we can't create `Animal` objects themselves, we can still reference subclasses of `Animal` (like `Mammal`) as `Animal` objects. That's the power of polymorphism.

Returning to the listing: after printing both `Mammal` objects, we call `getName()` on the object typed as an `Animal`. The `getName()` function exists in both `Mammal`, and its parent *super* class, `Animal`, recall. So what will it return, "Mammal", which is the type it truly is, or "Animal", the type it's being stored as?

The answer is "Mammal", because JavaFX functions are *virtual*, the subclass redefinition of `getName()` replaces the original in the parent class, even when the object is referenced by way of its parent type.

The last output line shows the `Reptile` inside the `Egg`. But why is its output different to the other `Reptile` object? Well, `layEgg()` never calls `born()`, so `birthDate` is null. And as `life` is not set either, the overridden initial value of 200 is used.

By the way, before we move on I do want to acknowledge to any women reading that I fully acknowledge childbirth is not as painless as creating a new object and calling a function on it! Likewise, similar sentiments to any reptiles who happen to be reading.

3.2.6 Multiple inheritance: compound and plain classes

JavaFX Script permits multiple inheritance, as alluded to in previous sections. Multiple inheritance is the ability to acquire variables and functions from more than one class at a time, through subclassing.

Java does not support multiple inheritance, but does allow a class to implement multiple interfaces in one go. Interfaces offer lightweight polymorphism, by supplying only the signature of a class without any code. The class must then fill out the code itself.

JavaFX's multiple inheritance creates a rift between itself and its big cousin. The Java Virtual Machine expects single inheritance, so the JavaFX Script compiler employs a cunning combination of classes and interfaces to make its multiple inheritance work. This is fine when dealing with pure JavaFX classes, but problems arise when adding Java classes into the mix. What happens, for example, when a JavaFX class extends a Java class?

We'll look at that in a moment, but for now let's familiarize ourselves with what multiple inheritance looks like in JavaFX Script:

Listing 3.12 Multiple inheritance

```
abstract class Motor                                                    A
{   var distance:Integer;                                               A
    function move(dir:Integer,dist:Integer) : Void                      A
    {   // Move in direction                                           A
        distance+=dist;                                                 A
    }                                                                   A
```

```
}                                                                    A

abstract class Weapon                                                B
{   var bullets:Integer;                                             B
    function fire(dir:Integer) : Void                                B
    {   if(bullets>0)                                                B
        {   // Fire weapon                                           B
            bullets--;                                               B
        }                                                            B
    }                                                                B
}                                                                    B

class Robot extends Motor,Weapon                                     C
{   var name:String;                                                 C
    function gameTurn() : Void                                       C
    {   var dir = 0;                                                 C
        // Calculate direction to move                               C
        move(dir,1);                                                 C
        fire(dir);                                                   C
    }                                                                C
}                                                                    C
```

A Class for movement
B Class for fighting
A Subclassing both the Motor and Weapon classes at once

The `Robot` class (see listing 3.12) is perhaps something we'd use in a video game. It inherits the capability to both move and fire a weapon. In our game there may be many objects which can fire but not move (fixed cannons), or move but not fire (a reconnaissance probe), so moving and firing are separate inheritable types. I couldn't be bothered with the full artificial intelligence needed to make our robot work, so I've added comments to describe what the code might do—fill the rest in with your imagination. As `Motor` and `Weapon` make no sense without subclassing, they're abstract classes.

As demonstrated, the `extends` construct cantake a comma separated list of classes. And that's really all there is to multiple inheritance—well, apart from one small detail. In order to co-operate with Java, JavaFX Script makes a distinction between classes it refers to as being *compound* and those it refers to as being *plain*. So what do these terms mean?

A plain class is any class which extends, directly or indirectly, a regular Java class. When we say a regular class we mean one which was written in Java—or any language which compiles to JVM compatible bytecode, other than JavaFX itself.

A compound class is any class entirely constructed from within JavaFX. Both itself and every class it inherits from, directly and indirectly, must be JavaFX classes.

So a plain class is one which either is a Java class itself, or inherits from a Java class. And a compound class is one which is a pure JavaFX Script class, with no Java classes in its inheritance. Now that we understand the terminology, here are the rules that define how the two types work together:

All JavaFX classes are compound by default.

A JavaFX class may directly inherit any number of compound classes, but only one plain

class.

Any JavaFX class which inherits a plain class, becomes a plain class.

So classes which live entirely within JavaFX can work happily with multiple inheritance, but classes which depend upon *outside* (non-JavaFX) classes are bound by Java's constraints of single inheritance. Whew, hopefully that's all clear!

3.2.7 Function types

Function types in JavaFX are incredibly useful. Not only do they provide a neat way of creating event handlers (see *anonymous functions*, later) but they allow us to plug bits of bespoke code into existing software algorithms. Functions in JavaFX Script are *first class objects*, meaning we can have variables of function type, and can pass functions into other functions as a parameters.

Listing 3.13 Function types

```
var func : function(:String):Boolean;
func = testFunc;
println( func("True") );
println( func("False") );

function testFunc(s:String):Boolean
{   return (s.equalsIgnoreCase("true"));
}
```

```
true
false
```

Listing 3.13 centers around the function at its tail, `testFunc()`, which accepts a `String` and returns a `Boolean`.

First we define a new variable, `func`, with a strange looking type. The variable will hold a reference to our function, so its type reflects the function signature. The keyword `function` is followed by the parameter list in parenthesis (variable names are optional), then a colon and the return type. In listing 3.13 the type is `function(:String):Boolean`, a function which accepts a single `String` and returns a `Boolean`. We can assign to this variable any function which matches that signature, and indeed in the very next line we do just that when we assign `testFunc` to `func`, using what looks like a standard variable assignment. We can now call `testFunc()` by way of our variable reference to it, which the code does twice just to prove it works.

Passing functions to other functions works along similar lines: the receiving function uses a function signature for its parameters, just like the variable above.

Listing 3.14 Passing functions as parameters to other functions

```
function manip(s:String ,                                    A
[CA]f:function(:String):String) : Void                       A
{   println("{s} = "+f(s));                                  A
}                                                            A
```

```
function m1(s:String) : String                                          B
{   s.toLowerCase();                                                    B
}                                                                       B
                                                                        B
function m2(s:String) : String                                          B
{   s.substring(0,4);                                                   B
}                                                                       B

manip("JavaFX" , m1);
manip("JavaFX" , m2);

JavaFX = javafx
JavaFX = Java
 A  Function with parameter function
 B  Functions we pass into manip()
```

The above cyan [CA]'s are code continuations, for wrapped lines.

The first function in listing 3.14, `manip()`, accepts two parameters and returns `Void`. The first parameter is of type `String`, and the second is of type `function(:String):String`, which in plain English translates as a function which accepts a `String` and returns a `String`. Fortunately we happen have two such functions, `m1()` and `m2()`, both accept and return a `String`, performing basic manipulations in between. We call `manip()` twice, passing in a `String` and one of our functions each time. The `manip()` function invokes the parameter function with said `String` and prints the result. A simple example, perhaps, yet one that adequately demonstrates the effect.

Being able to pass functions around like this is quite a power feature. Imagine a list capable of being sorted or filtered by a plug-in function, for example. But this isn't the end of our discussion. Next section we continue to look at functions, this time with a twist.

3.2.8 Anonymous functions

We've just seen how we can pass functions into other functions, and assign them to variables, but what applications does this have? The most obvious one is callbacks—or event handlers as they're more commonly known in the Java world.

In a GUI environment we frequently need to respond to events originating from the user—when they click a button or slide a scrollbar we need to know. Typically we register a piece of code with the class generating the event, to be called when said event happens. JavaFX Script's function types, with their ability to point at code, fit the bill perfectly.

Having to create complete functions for each event handler is a pain, especially as in many cases they're only used in one place. If only there was a short-cut syntax for one-time function creation. Well, unsurprisingly, there is.

Listing 3.15 Anonymous functions

```
import java.io.File;

class FileSystemWalker
{   var root:String;
```

```
    var extension:String;
    var action:function(:File):Void;

    function go() { walk(new File(root)); }

    function walk(dir:File) : Void
    {   var files:File[] = dir.listFiles();
        for(f:File in files)
        {   if(f.isDirectory())
            {   walk(f);
            }
            else if(f.getName().endsWith(extension))
            {   action(f);
            }
        }
    }
}

var walker = FileSystemWalker
{   root: FX.getArguments()[0];
    extension: ".png";
    action: function(f:File)                                          A
    {   println("Found {f.getName()}");                               A
    }                                                                 A
};
walker.go();
```

A Anonymous function

The class `FileSystemWalker` (listing 3.15) has three variables and two functions. One of the variables is a function type, called `action`, which can point to functions of type `function(:File):Void`—or, in plain English, any function which accepts a `java.io.File` object and returns nothing.

The most important function is `walk()`, which recursively walks a directory tree looking for files which end with the desired `extension`, calling whichever function has been assigned to `action` for each match, passing said file in as a parameter. The other function, `go()`, merely acts as a convenience to kick-start the recursive process from a given `root` directory. So far, nothing new! But it starts to get interesting when we see how `walker`, the object of type `FileSystemWalker`, is constructed.

In its declaration `walker` assigns the `root` directory to the first parameter passed in on the command line—so when you run the code make sure you nominate a directory! (The `FX.getArguments()` function is how we get at the command line arguments, by the way.) The `extension` is set to PNG files, so `walk()` will only act on filenames with that ending.

Look at the way `action` is assigned: rather than pointing to a function elsewhere, the code merely defines a nameless (anonymous) function of the required type right there as part of the assignment. This is an *anonymous function*, a define-and-forget piece of code assigned to a variable of function type. It allows us to plug short snippets of code into existing classes without the inconvenience of having to fill up our scripts with fully fleshed out functions. Ideal for quick and easy event handling.

3.2.9 Access modifiers (package, protected, public, public-read, public-init)

We round off our look at classes by examining how to keep secrets. Classes encapsulate related variables and functions into self contained objects, but an object becomes truly self contained when it can lock out third parties from its implementation detail.

JavaFX's access modifiers are tailored to suit the JavaFX Script language, and its declarative syntax. Access modifiers can be applied to script members (functions and variables at script level), instance members (functions and variables inside a class), and classes themselves. They cannot be used with, indeed make no sense for, local variables inside functions. (See listing 3.3 for an example of different types of variable.)

There are four basic modes of visibility in JavaFX Script, outlined in table 3.2:

Table 3.1 Basic access modifiers

Modifier keyword	Visibility effect
(default)	Visible only within the enclosing script. This default mode (with no associated keyword) is the least visible of all access modes.
`package`	Visible within the enclosing script, and any script or class within the same package.
`protected`	Visible within the enclosing script, any script or class within the same package, and subclasses from other packages. This modifier only works with class members, not script members or the class itself.
`public`	Visible to anyone, anywhere.

There are two additive access modifiers, which may be combined with the four modes in table 3.1 above–modifiers to the modifiers, if you like. They are designed to compliment JavaFX Script's declarative (object literal) syntax. As such, they only apply to `var` variables (capable of being modified), and cannot be used with functions, classes, or any `def` variables. Table 3.2, below, details them.

Tables 3.2 Additive access modifiers

Modifier keyword	Visibility effect
`public-read`	Adds public read access to the basic mode.
`public-init`	Adds public read access and object literal write access to the basic mode.

The `public-read` modifier grants readability to a variable, while writing is still controlled by its basic mode. The `public-init` modifier also grants public writing, but only during object declaration. Writing at other times is still controlled by the basic mode.

Each modifier solves a particular problem, so the clearest way to explain they're use is with a task centric FAQ, like the one up next:

- Q: I've written a script/class, and don't want other scripts messing with my functions or variables, as I might change them at a later date. Can I do this?
  A: Stick with the default access mode. It gives you complete freedom with your variables and functions because no other script can interact with them.
- Q: I'm writing a package. Some functions and variables need to be accessible across scripts and classes of the package, but I don't want other programmers getting access to them. Is this possible?
  A: Sure, the `package` *access modifier will do that for you.*
- Q: Some of my class's functions and variables would be useful to authors of subclasses, but I don't want to open them up to the world. How is this done?
  A: Check out the `protected` *access modifier, it grants package visibility, plus any subclasses from outside the package.*
- Q: I have a class with some variables I'd like to make readable by everyone, but I still want to control write access to them. Can JavaFX Script do this?
  A: Indeed! Just combine `public-read` *with one of the four basic modes.*
- Q: I'd like to control writing to my instance variables, except when the instance is first created. Possible?
  A: Funny you should ask. Just add `public-init` *to one of the four basic modes, and your variables will become public writable when used from an object literal.*
- Q: So, why can't I use these additive modifiers with def variables?
  A: Common sense. A `public-read def` *would be the same as a* `public def`*, and a* `public-init def` *would be rather pointless.*

Enough questions, let's consider some actual source code.

Listing 3.16 Access modifiers on a class

```
package jfxia.chapter3.access;

public class AccessTest
{   var sDefault:String;                                                    A
    package var sPackage:String;                                            A
    protected var sProtected:String;                                        A
    public var sPublic:String;                                              A

    public-read var sPublicReadDefault:String;                              B
    public-read package var sPublicReadPackage:String;                      B
    public-init protected var sPublicInitProtected:String;                  B

    init
    {   println("sDefault = {this.sDefault}");
        println("sPackage = {this.sPackage}");
        println("sProtected = {this.sProtected}");
        println("sPublic = {this.sPublic}");
        println("sPublicReadDefault = "
```

```
            "{this.sPublicReadDefault}");
        println("sPublicReadPackage = "
            "{this.sPublicReadPackage}");
        println("sPublicInitProtected = "
            "{this.sPublicInitProtected}");
    }
}
```

A Basic modes
B Additive modes

Listing 3.16 shows a class with instance variables displaying various types of access visibility. Note that the class is in package `jfxia.chapter3.access`. To test it we need some further sample code.

Listing 3.17 Testing access modifiers

```
package jfxia.chapter3;
import jfxia.chapter3.access.AccessTest;


def a:AccessTest = AccessTest
{   // ** sDefault has script only (default) access
    //sDefault: "set";

    // ** sPackage is not public; cannot be accessed
[CA]from outside package
    //sPackage: "set";

    // ** sProtected has protected access
    //sProtected: "set";

    sPublic: "set";                                                    A

    // ** sPublicReadDefault has script only (default)
[CA]initialization access
    //sPublicReadDefault: "set";

    // ** sPublicReadPackage has package initialization
[CA]access
    //sPublicReadPackage: "set";

    sPublicInitProtected: "set";                                       B
};

// ** sPublicInitProtected has protected write access
//a.sPublicInitProtected = "set2";                                     C

def str:String = a.sPublicReadDefault;                                 D

sDefault =
sPackage =
sProtected =
sPublic = set
sPublicReadDefault =
sPublicReadPackage =
sPublicInitProtected = set
```

A Always works

B Works during declaration
C Fails outside declaration
D Read is okay

The above cyan [CA]'s are code continuations, for wrapped lines.

Listing 3.17 tests the `AccessTest` class we saw in listing 3.16. It lives in a different package to `AccessTest`, so we can expect all manner of access problems. The script builds an instance of `AccessTest`, attempting to set each of its instance members. The lines that fail have been commented out, with the compilation error listed above.

Of the seven variables, only two are successfully accessible during the object's declaration, one of which is the `public` variable allowing total unhindered access.

Keen eyes will have spotted the `protected` variable cannot be assigned, but its `public-init protected` cousin can. The `public-init` modifier grants write access only during initialization–which is why a second assignment, outside the object literal, fails.

Also, note how the `public-read 'default'` variable has become read-only outside of its class.

So that's it for access modifiers. By sensibly choosing access modes we can create effective components, allowing other programmers to interact with them through clearly defined means, while protecting their inner implementation detail.

And so ends our discussion of classes. Next section we begin studying familiar code constructs like conditions and loops, but with an expression language twist.

3.3 Flow control, using conditions

Conditions are a standard part of all programming languages. Without them we'd have straight line code, doing the same thing every time, with zero regard for user input or other runtime stimuli. This would cut dramatically the number of bugs in our code, but would ever-so-slightly rendering all software completely useless.

JavaFX Script's conditions behave in a not too dissimilar fashion to other languages, but the expression syntax permits one or two interesting tricks.

Use your imagination

The demonstration conditions in the sections below are somewhat contrived. Hard coded values mean the same path is always followed each time the code is run. I *could* have written each example to accept some runtime 'variable' (an external factor, not determinable at compile-time) such that each path could be exercised. While this would add an element of 'real-world-ness', it would also make the code much longer, without adding any demonstration value. I think it goes without saying, I consider readers of this book to be intelligent enough to study each example and dry-run in their heads how different data would activate the various paths through the code.

3.3.1 Basic conditions (if, else)

We'll begin with a basic example, to kick things off:

Listing 3.18 Conditions

```
var someValue = 99;

if(someValue==99)
{   println("Equals 99");
}

if(someValue >= 100)
{   println("100 or over");
}
else
{   println("Less than 100");
}

if(someValue < 0)
{   println("Negative");
}
else if(someValue > 0)
{   println("Positive");
}
else
{   println("Zero");
}
```

```
Equals 99
Less than 100
Positive
```

There are three condition examples in listing 3.18, all depending upon the variable `someValue`. The first is a straight `if` block, its code is either run or it is not. The second adds an `else` block, which will run if its associated condition is false. The third adds another condition block, which is tested only if the first condition is false.

3.3.2. Conditions as expressions

So JavaFX's if/elseif/else construct is the same as countless other programming languages, but you'll recall mention of “interesting tricks”—let's have have a look at an example. Check out listing 3.19, below.

Listing 3.19 Conditions as expressions

```
var negValue = -1;
var sign = if(negValue < 0) { "Negative"; }
    else if(negValue > 0) { "Positive"; }
    else { "Zero"; }
println("sign = {sign}");
```

```
sign = Negative
```

Your eyes do not deceive. We are indeed assigning from a condition!

The variable `sign` takes its value directly from the result of the condition which follows. It will either acquire the value “Positive”, “Negative or “Zero” depending upon the outcome of the condition. How is this happening? Let's all chant the mantra together, shall we? “JavaFX Script is an expression language, JavaFX Script is an expression language...!”

JavaFX's conditions give out a result, and as such can be used on the right hand side of an assignment, or as part of a bind, or any other situation in which a result is expected. Now, perhaps, you understand why we were able to use conditions directly inside formatted strings, or to update bound variables.

3.3.3 Ternary expressions, and beyond

Let's expand on the above notion. In other languages there's a concept of a ternary expression, which consists of a condition followed by two results, the first is returned if the condition is true, the second if it is false. We can achieve the same thing via JavaFX Script's if/else, as shown below in listing 3.20:

Listing 3.20 Ternary expressions

```
import java.lang.System;

var asHex = true;
System.out.printf
(   if(asHex) "Hex:%04x%n" else "Dec:%d%n" ,
    12345
);
```

```
Hex:3039
```

Depending upon the value of `isHex` either the first or the second formatting string will be applied to the number `12345`. You'll note in the lack of curly braces and the absence of a closing semi-colon in each block of the if/else. When used in a ternary sense, with each block of the `if` construct housing a single expression, we can't put semi-colons inside the blocks—the compiler would assume said block houses multiple expressions, which would not fit the ternary syntax. Any semi-colon should come at the end of the entire if/else construct.

Let's try something a little more ambitious:

Listing 3.21 Beyond ternary expressions

```
var mode = 2;
println
(   if(mode==0) "Yellow alert"
    else if(mode==1) "Orange alert"
    else if(mode==2) "Mauve alert"
    else "Red alert"
);
```

```
Mauve alert
```

Listing 3.21 shows the power of conditions being expressions. What amounts to a *switch* construct is actually directly providing the parameter for a Java method call, without setting a variable first or wrapping itself in a function. Naturally, because of our hard coded `mode`

the alert will always be set to mauve (besides, as every Red Dwarf fan knows, red alert would require changing the light bulb!)

This idea of conditions having results is a powerful one, so let's push it to its logical, or should that be illogical, conclusion:

Listing 3.22 Condition expressions taken to an extreme

```
import java.lang.System;

var rand = (System.currentTimeMillis() as Integer) mod 2;
var flag:Boolean = (rand == 0);
var ambiguous = if(flag) 99 else "Hello";

println("{rand}: flag={flag}, ambiguous={ambiguous} "
     "({ambiguous.getClass()})");
```

```
0: flag=true, ambiguous=99 (class java.lang.Integer )                    A
1: flag=false, ambiguous=Hello (class java.lang.String)                  A
```

A Two different executions

Admittedly, when I first wrote the code above (listing 3.22), I expected a compiler error. None was forthcoming, however. This time we don't use hard coded values, but a weak pseudo random event to feed the decision logic. First we use a Java API method to get the POSIX time in milliseconds (the number of milliseconds elapsed since midnight, 1st January 1970), setting a variable called `flag` such that sometimes when we run the code the result will be true, and other times false. Another variable, `ambiguous`, is then set depending upon `flag`—if true it will be assigned an `Integer`, and if false a `String`.

So the type of `ambiguous` is dependent on the path the code takes—I'm not sure I like this (and would strongly urge you not to make use of such ambiguous typing in your own code) but JFX seems to handle it without complaint.

Anyway, with that rather dangerous example, we'll leave conditions behind and move on to something else—loops.

3.4 Sequence based loops

Loops are another staple of programming, allowing us to repeatedly execute a given section of code until a condition is met. In JavaFX Script loops can be tied firmly to sequences, and like conditions they hold a trick or two when it comes to being treated as expressions.

3.4.1 Basic sequence loops (for)

Let us begin with a basic example, by way of introducing the syntax:

Listing 3.23 Basic for loops

```
for(a in [1..3])
{   for(b in [1..3])
    {   println("{a} x {b} = {a*b}");
    }
}
```

```
1 x 1 = 1
```

```
1 x 2 = 2
1 x 3 = 3
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
```

If you ever forget your three times table, now you have a convenient JavaFX program to print it for you—two loops, one inside the other, with a `println()` at the center of it all. Note how in listing 3.23 we tie the loop to a sequence, defining its range, then pulling each element out into the loop variable.

3.4.2 For loops as expressions (indexof)

Now for something which exploits the expression language facilities:

Listing 3.24 Sequence creation using for expressions

```
var cards =
    for(str in ["A",[2..10],"J","Q","K"])
        str.toString();
println(cards.toString());

cards =
    for(str in ["A",[2..10],"J","Q","K"])
        if(indexof str < 10) null else str.toString();
println(cards.toString());
```

```
[ A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K ]
[ J, Q, K ]
```

A `for` loop returns a sequence, and listing 3.24 shows us exploiting that fact by using a loop to construct a sequence of strings using `toString()` over a mixture of strings and integers. Each pass through the loop an element is plucked from the source sequence, converted into a string, and added to the destination sequence, `cards`. The loop variable becomes a `String` or an `Integer` depending upon the element in the source sequence. Fortunately all variables in JavaFX Script inherit `toString()`.

If you ever need to know the index of an element during a `for` expression, `indexof` is your friend. The first element is zero, the second is one, and so on. Null values are ignored when building sequences, so not every iteration through the loop need extend the sequence contents. The second part of listing 3.24 shows both these feature in action.

3.4.3 Rolling nested loops into one expression

The loop syntax gives us an easy way to create loops within loops, allowing us, for example, to drill down to access sequences inside objects within sequences. In the following example (listing 3.25) we use a sequence of `SoccerTeam` objects, each containing a sequence of player names.

Listing 3.25 Nested loops within one for expression

```
class SoccerTeam                                                    A
{   public-init var name: String;                                   A
    public-init var players: String[];                              A
}                                                                   A

var fiveAsideLeague:SoccerTeam[] =
[   SoccerTeam                                                      B
    {   name: "Java United"                                         B
        players: [ "Smith","Jones","Brown",                         B
            "Johnson","Schwartz" ]                                  B
    } ,                                                             B
    SoccerTeam                                                      C
    {   name: ".Net Rovers"                                         C
        players: [ "Davis","Taylor","Booth",                        C
            "Williams","Ballmer" ]                                  C
    }                                                               C
];

for(t in fiveAsideLeague, p in t.players)                           D
{   println("{t.name}: {p}");                                       D
}                                                                   D
```

```
Java United: Smith
Java United: Jones
Java United: Brown
Java United: Johnson
Java United: Schwartz
.Net Rovers: Davis
.Net Rovers: Taylor
.Net Rovers: Booth
.Net Rovers: Williams
.Net Rovers: Ballmer
```

A A soccer team class
B First soccer team object
C Second soccer team object
D Print players within teams

First we define our team class, then we create a sequence of two five-a-side teams, each team with a name and five players. The meat of the example comes at the end, when we use a single `for` statement to print each player from each team. Each level of the loop is separated by a comma. We have the outer part which walks over the teams, and we have the inner part which walks over each player within each team. It means we can unroll the whole structure with just one `for` expression, instead of multiple nested expressions.

3.4.4 Controlling flow within for loops (break, continue)

JavaFX Script supports both the `continue` and `break` functionality of other languages in its for loops. Continues skip on to the next iteration of the loop without executing the remainder of any code in the body. Breaks terminate the loop immediately. Unlike with other languages, JavaFX Script breaks do not support a label to point to a specific for loop to break out of.

Listing 3.26 provides an example:

Listing 3.26 Flow control within for, with break and continue

```
var total=0;
for(i in [0..50])
{   if(i<5) { continue; }
    else if (i>10) { break; }
    total+=i;
}
println(total);
```

```
45
```

The loop runs, supposedly, from 0 to 50. However we ignore the first five passes through by using a `continue`, and we force the loop to terminate with a `break` when it gets beyond 10. In effect `total` is only updated for 5 to 10, which explains the result (5+6+7+8+9+10 = 45).

3.4.5 Filtering for expressions (where)

There's one final trick we should cover when it comes to loops: applying a filter to selectively pull out only the elements of the source sequence we want.

Listing 3.27 Filtered for expression

```
var divisibleBy7 =
    for(i in [0..50] where (i mod 7)==0) i;
println(divisibleBy7.toString());
```

```
[ 0, 7, 14, 21, 28, 35, 42, 49 ]
```

The loop in listing 3.27 runs over each element in a sequence from 0 to 50, but the added `where` clause filters out any loop value which isn't evenly divisible by 7. The result is the sequence, `divisibleBy7`, who's contents we print.

So that's it for sequence based loops, at least for now. Later we'll touch briefly on sequences again, when we visit triggers. In the next section we'll consider a more conventional type of loop.

3.5 Repeating code, with while loops (while, break, continue)

As well as sequence-centric for loops, JavaFX Script supports while loops, in a similar fashion to other languages. The syntax is fairly simple, so we begin, as ever, with an example:

Listing 3.28 Basic while loops

```
var i=0;
var total=0;
while(i<10)
{   total+=i;
    i++;
}
println(total);

i=0;
total=0;
while(i<50)
{   if(i<5) { i++; continue; }
    else if (i>10) { break; }
```

```
    total+=i;
    i++;
}
println(total);
```

```
45
45
```

Two while loops in listing 3.28, the first a simple loop, the second involving some break and continue logic. To create a while loop we use the keyword `while`, followed by a terminating condition in parenthesis, and the body of the loop in curly braces. The original loop above walks over the numbers 0 to 9, by way of the variable `i`, totaling each loop value as it goes. The result of totaling all the values 0 to 9 is 45.

The second loop performs a similar feat, walking over the values 0 to 50—or does it? The `continue` keyword is triggered for all values under 5, and the `break` statement is triggered when the loop exceeds 10. The former will cause the body of the loop to be skipped, jumping straight to the next iteration, while the later causes the loop to be aborted. The result is that only the values 5 to 10 are totaled, also giving the answer 45.

Unlike some languages, JavaFX Script `break` statements do not support a label pointing to which loop (of several nested) to break out of.

3.6 Act on variable and sequence changes, using triggers

Triggers allow us to assign some code to run when a given variable is modified. It's a simple, yet powerful, feature which can greatly aid us in creating sophisticated code which reacts to data change.

3.6.1 Single value triggers (on replace)

Here's some example code to demonstrate a simple trigger in action:

Listing 3.29 Trigger on variable change

```
class TestTrigger
{   var current = 99
        on replace oldVal = newVal                                    A
        {   previous = oldVal;                                        A
        };                                                            A
    var previous = 0;

    override function toString() : String
    {   "current={current} previous={previous}";
    }
}

var trig1 = TestTrigger {};
println(trig1);
trig1.current = 7;
println(trig1);
trig1.current = -8;
println(trig1);
```

```
current=99 previous=0
```

```
current=7 previous=99
current=-8 previous=7
```

A Runs when 'current' updated

As written in listing 3.29 the class `TestTrigger` has two variables, the first of which has a trigger attached to it. Triggers are added to the end of a variable declaration, using the keyword phrase `on replace`, followed by a variable to hold the current value, an equals, and a variable to hold the replacement value. In the above example `oldVal` will contain the existing value of `current` when the trigger is activated, and `newVal` will contain the updated value. We use the old value to populate a second variable, `previous`, ensuring it is always one step behind `current`.

Note: the equals sign used as part of the `on replace` construct is just a separator, it's not an assignment. I suppose `oldVal` and `newVal` could have been separated by a comma, but the language designers presumably thought equals was more intuitive.

The code block is called **after** the variable is updated, so `newVal` is already in place when our code starts. Actually, we could have just read `current` instead of `newVal`.

Listing 3.30 Shorter trigger syntax

```
var onRep1:Integer = 0 on replace                                    A
{   println("onRep1: {onRep1}");
}
var onRep2:Integer = 5 on replace oldVal                             B
{   println("onRep2: {oldVal} => {onRep2}");
}
onRep1 = 99;
onRep2++;
onRep2--;

onRep1: 0                                                            C
onRep2: 0 => 5                                                       C
onRep1: 99
onRep2: 5 => 6
onRep2: 6 => 5
```

A No old or new value
B Old value only
C Initialization

We can use an abbreviated syntax which misses off the new value, and the old value too, if we want. Listing 3.30 shows both variants.

3.6.2 Sequence triggers (on replace [..])

As you'd expect, we can also assign a trigger to a sequence. To do this we need to also tap into not only the existing and replacement values, but the range of the sequence which is being affected. Fortunately we can use a trigger itself to demonstrate how it works:

Listing 3.31 Triggers on a sequence

```
var seq1 = [1..3]
    on replace oldVal[firstIdx..lastIdx] = newVal
    {   println
        (   "Changing [{firstIdx}..{lastIdx}] from "
```

```
            "{oldVal.toString()} to "
            "{newVal.toString()}"
        );
    };
println("Inserts");
insert 4 into seq1;
insert 0 before seq1[0];
insert [98,99] after seq1[2];
println("Deletes");
delete seq1[0];
delete seq1[0..2];
delete seq1;
println("Assign then reverse");
seq1 = [1..3];
seq1 = reverse seq1;
```

```
Changing [0..-1] from [ ] to [ 1, 2, 3 ]
Inserts
Changing [3..2] from [ 1, 2, 3 ] to [ 4 ]
Changing [0..-1] from [ 1, 2, 3, 4 ] to [ 0 ]
Changing [3..2] from [ 0, 1, 2, 3, 4 ] to [ 98, 99 ]
Deletes
Changing [0..0] from [ 0, 1, 2, 98, 99, 3, 4 ] to [ ]
Changing [0..2] from [ 1, 2, 98, 99, 3, 4 ] to [ ]
Changing [0..2] from [ 99, 3, 4 ] to [ ]
Assign then reverse
Changing [0..-1] from [ ] to [ 1, 2, 3 ]
Changing [0..2] from [ 1, 2, 3 ] to [ 3, 2, 1 ]
```

The output from listing 3.31 tells the story of how the trigger was used. The `on replace` syntax has been complimented by a couple of new variables boxed in square brackets. Don't be confused, this extension to `on replace` isn't actually a sequence itself, the language designers just borrowed the familiar syntax to make it look more intuitive.

Watch carefully how the two values change as we perform various sequence operations.

When the sequence is first created a trigger call is made adding the initial values at index 0. Then we see three further insert operations. Each time `oldVal` is set to the pre-insert contents, `newVal` is set to the contents being added, `firstIdx` is the index to which the new values will be added, and `lastIdx` is one behind `firstIdx` (it has little meaning during an insert, so just gets an arbitrary value).

We next see three delete operations. Again `oldVal` is the current content of the sequence before the operation, `newVal` is an empty sequence (there's no new values in a delete, obviously), and `firstIdx` and `lastIdx` describe the index range of the elements being removed.

Finally we re-populate the sequence with some fresh data, causing an insert trigger, then reverse the sequence to cause a mass replacement. Note how during the reverse the `firstIdx` and `lastIdx` values actually expresses the elements being modified, unlike with an insert where only the lower index is used.

Triggers can be really useful in certain circumstances, but we should avoid temptation to abuse them; the last thing we want is code which is hard to understand and a nightmare to

debug. And talking of code which doesn't do what we think it should, next section we look at exceptions (talk about a slick segue!)

3.7 Trap problems using exceptions (try, catch, any, finally)

To misquote the famous cliché, "stuff happens!" And when it happens we need some way of knowing about it. Exceptions give us a way to assign a block of code to be run when a problem occurs, or to signal a problem within our own code to outside code which may be using our API. As always, we begin with an example:

Listing 3.31 Exception handling

```
import java.lang.NullPointerException;
import java.io.IOException;

var key = 0;
try
{   println(doSomething());
}
catch(ex:IOException)
{   println("ERROR reading data {ex}")
}
catch(any)
{   println("ERROR unknown fault");
}
finally
{   println("This always runs");
}

function doSomething() : String
{   if(key==1)
    {   throw new IOException("Data corrupt");
    }
    else if(key==2)
    {   throw new NullPointerException();
    }
    "No problems!";
}

No problems!                                                    A
This always runs                                                A

ERROR reading data java.io.IOException: Data corrupt            B
This always runs                                                B

ERROR unknown fault                                             C
This always runs                                                C
 A  key = 0
 B  key = 1
 C  key = 2
```

The code above (listing 3.31) hinges on the value of `key`, determining which exceptions may be thrown. The example is a little contrived, but it's compact and demonstrates the mechanics of exceptions perfectly well. The catch blocks are executed if the `doSomething()` function throws an exception. The first block will be activated if the

function throws an `IOException`. The second uses the `any` keyword to trap other exception which might be thrown. And lastly the `finally` block will always be executed, regardless of whether an exception occurred or not.

The results, in bold, show the code being run with different values for `key`. First we have a clean run with no exceptions; the function returns normally, the results are printed, and the `finally` block is run. Second we have a (simulated) IO failure, causing the function to abort by throwing an `IOException`, which is trapped by our first `catch` block, and again the `finally` block runs at the close. In the third run we cause the function to abort with a `NullPointerException`, triggering the catch-all exception handler, and once again the `finally` block runs at the close.

The finally block is a useful device for cleaning up after a piece of code. For example, we should close files properly when leaving a section of code. To avoid identical code in multiple places the `finally` block should be used. Its contents will run no matter how the `try` block exits. We can even use finally blocks without catch blocks, keeping code clean by putting must-run terminating code in a single place.

3.8 Summary

JavaFX Script may seem a little quirky in places to someone coming to it fresh, but its quirks all generally seem to make sense. It's an expression language syntax might seem a little bizarre at first—assigning from `if` and `for` blocks takes some getting used to—but it permits code and data structures to be interwoven seamlessly. Binding and triggers allow us to define relationships between variables, and code to run when those variables change. But more importantly they permits us to put such code right next to the variables they relate to—rather than in some disparate block or source file, miles away from where the variable is actually defined.

We've covered so much over the last few dozen pages, I wouldn't be at all surprised if many of you felt your heads where spinning. What we need is a nice, gentle, project to get us started. Something fun, yet with enough challenge to allow us to practice some of the unique JavaFX Script features we've just finished learning about.

In the next chapter we're not going to jump straight in with animations and swish user interfaces, instead we're keeping it nice and simple by developing a Swing-like application—a number puzzle game like the ones found in many newspapers. So, take a short break, brew yourself a fresh cup of coffee, and I'll see you in the next chapter...

4

Swing by numbers

We've had to take in an awful lot in the last couple of chapters—an entirely new language no less! I know many of you will be eager to dive straight into creating media rich applications, but we need to learn to walk before we can run. JavaFX Script gives us a lot of powerful tools for writing great software, but all we've seen thus far is a few abstract examples.

So for this, our first project, we won't be developing any flashy visuals or clever animations. Be patient. Instead we need to start putting all the stuff we crammed into our brains over the last few dozen pages to good use. A common paradigm in user interface software is Model/View/Controller, where data and UI are separate, interacting by posting updates to each other. The *model* is the data, while the *view*/*controller* is the display and its input. We're going to develop a data class, and a corresponding UI, to see how the language features of JavaFX Script allow us to bind them together (pun only partially intended). But first we need to decide on a simple project to practice on; something fun, yet informative. I think I know just the thing!

We're going to develop a version of the simple, yet addictive, number puzzle game found in countless newspapers and magazines around the World. If you've never encountered such puzzles before, take a look at figure 4.1, below.

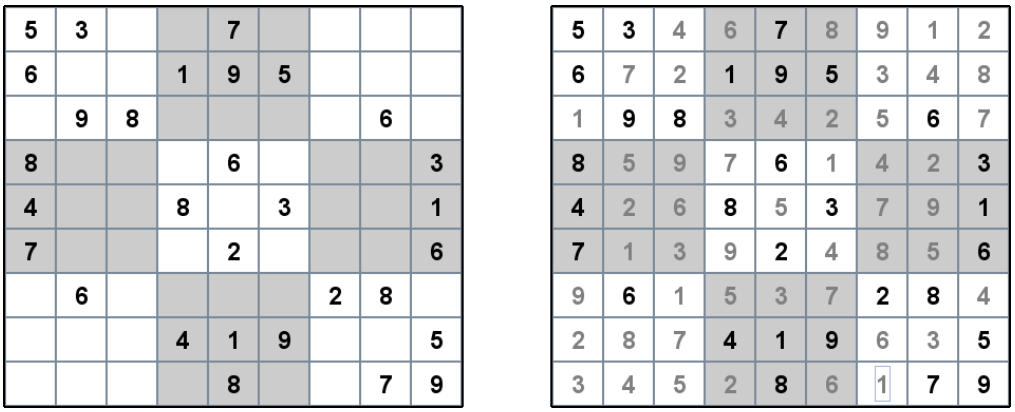


Figure 4.1 A number puzzle grid, shown both empty and recently completed (note the UI focus highlight on the bottom row.)

The general idea is to fill out the missing cells in a grid with unique numbers (in a standard puzzle, 1 to 9) in each *row*, each *column* and each *box*. A successful solution is a grid completely filled out, without duplicates in any row, column, or box.

The number puzzle

Number puzzles like the one we're developing have been published in magazines and newspapers since the late 19th century. By the time of the First World War, however, their popularity had waned and slowly they fell into obscurity. In the late 1970s the puzzle was re-invented, legend has it, by an American puzzle author named Howard Garns, and eventually found its way to Japan were it gained the title “Sudoku”. It took another twenty-five years, however, for the puzzle to become popular in the West. Its inclusion in the UK's Sunday Times newspaper was an overnight success, and from there it has gone on to create addicts around the World.

By far the most common puzzle format is a basic 9 x 9 grid, giving us: nine rows of nine cells each, nine columns of nine cells each, and nine 3 x 3 boxes of nine cells each. At the start of the puzzle a grid is presented with some of the numbers already in place. The player must fill in the missing cells using only the numbers 1 to 9, such that all twenty-seven *groups* (nine rows, nine columns, nine boxes) contain only one occurrence of each number.

We'll be using some of JavaFX's Swing components in our application. JavaFX's standard APIs house many classes which wrap the main Swing UI components. For those of you who have developed with Swing through Java in the past, this will be a real eye opener. You'll see firsthand how the same user interfaces you previously created with reams and reams of Java code can be constructed with relatively terse declarative JavaFX code. For those of you

who haven't encountered the delights of Swing, this will just be a gentle introduction to creating traditional Uis with the power tools JFX provides. Either way, hopefully we'll have fun!

This project is not a comprehensive Swing tutorial. Swing is a huge and very complex beast, with books the size of telephone directories published about it. JavaFX only provides direct support (JFX wrappers) for a handful of core Swing components, although the whole of Swing can be used as Java objects, of course. The project is primarily about showing how a Swing-like UI can be constructed, quickly and cleanly, using JavaFX Script.

WARNING: SWING WILL SOON BE SLUNG

In this chapter we're using Swing wrappers to develop our UI. I've done this for two reasons: firstly it gives existing Swing developers a direct comparison of developing in JavaFX Script versus Java, and secondly, JFX's own *native* UI library (housed in `javafx.scene.control`) is not ready as of the JavaFX 1.0 release being used to write this book. In future releases of JavaFX the Swing library will be pushed firmly into the background, as JavaFX gets its own controls API. For now, though, Swing will suit our purposes.

Enough about Swing, what about our number puzzle? I don't know if you've ever noticed, but often the simpler the idea, the harder it is to capture in words alone. Sometimes it's far quicker to learn by seeing something in action. Our number puzzle uses blissfully simple rules, yet it's hard to describe in the abstract. So to avoid confusion we need to agree on a few basic terms before we proceed:

- The grid is playing area on which the puzzle is played.
- A row is horizontal line of cells in the grid.
- A column is a vertical line of cells in the grid.
- A box is a sub-grid within the grid.
- A group is any segment of the grid which must contain unique numbers (all rows, columns and boxes.)
- A position is a single cell within a group.

The various elements are demonstrated by figure 4.2, below.

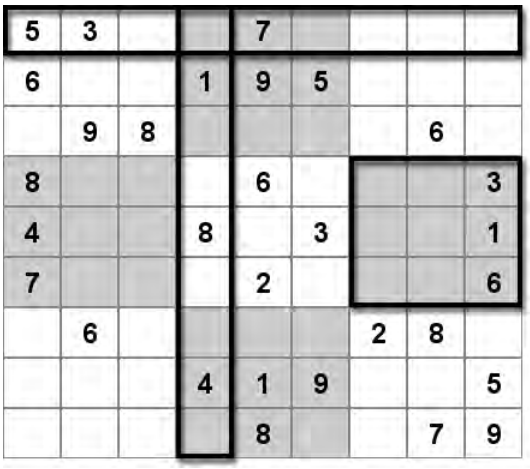


Figure 4.2 Groups are rows, columns or boxes within the grid, which must hold unique values

Rather than throw everything at you at once, we're going to develop the application piece by piece, building it up slowly as we go. With each stage we'll add in a little bit more functionality, using the languages features we learned about in the previous two chapters, and you'll see how they can be employed to make our application work.

Right then, compiler to the ready, let's begin...

4.1 Swing time: puzzle, version 1

What is Swing? Well, back when Java first entered the market, it's official user interface API was known as AWT (Abstract Window Toolkit), a library which sought to smooth over the differences between the various *native* GUI toolkits on Windows, the Mac, Linux, and any other desktop environment it was ported to. AWT wrapped the operating system's own native GUI widgets (buttons, scrollbars, text areas, etc.) to create a consistent environment across all platforms. Yet because of this it came under fire as being a lowest-common-denominator solution, a subset of only the features available on all platforms. So in answer to this criticism a more powerful alternative was developed: Swing!

Swing sits atop AWT, but uses only its most low-level features—pixel pushing and keyboard/mouse input mainly—to deliver an entirely Java based library of UI widgets. Swing is a large and very powerful creature, quite possibly one of the most powerful (certainly one of the most complex) UI toolkits ever created. In this project, fortunately, we'll be looking at just a small part of it.

As we're developing our puzzle bit by bit, in version one we won't expect to have a working game. We're just laying the foundations for what's to come.

What's in a name?

The generic name for a user interface control differs from system to system, and from toolkit to toolkit. In the old Motif (X / X-Windows) toolkit they were called “widgets”, Windows uses the boring term “controls”, Java AWT/Swing calls them by the rather bland name “components”, and I seem to recall the Amiga called them (bizarrely!) “gadgets”. Looks like nobody can agree!

I like “widget”! Why? Well, “control” and “component” are just too non-specific, and open to ambiguity. Widget isn't likely to crop up in any other context when discussing user interface programming. Plus it's just a much more quirky word, far more interesting a sound than the sterile, unimaginative, “component”.

4.1.1 Our initial puzzle data class

We need to start somewhere, so here's some basic code which will get the ball rolling, defining the data we need to represent our puzzle. Listing 4.1, below, is our initial shot at a main puzzle grid class, but as you'll see it's far from finished.

Listing 4.1 PuzzleGrid.fx (version 1)

```
package jfxia.chapter4;

package class PuzzleGrid
{   public-init var boxDim:Integer = 3;
    public-read def gridDim:Integer = bind boxDim*boxDim;
    public-read def gridSize:Integer = bind gridDim*gridDim;

    package var grid:Integer[] =
        for(a in [0..<gridSize]) { 0 }
}
```

So far all we have is four variables, which do the following:

- The `boxDim` variable is the dimension of the boxes inside the main grid. Boxes are square, so we don't need separate width and height dimensions.
- The `gridDim` variable holds the width and height of the game grid. The grid is square, so we don't need to hold separate values for both dimensions. This value is always the square of the `boxDim` variable, so we utilize a bind to ensure they remain consistent.
- The `gridSize` variable is a convenience for when we need to know the total number of cells in the grid. You'll note we're using a binding here too, when `gridDim` is changed `gridSize` will automatically be updated.
- Finally we have the `grid` itself, represented as a sequence of `Integer` values. We'll create an initial, default, grid with all zeros for now using a `for` loop.

So far, so good. We've defined our basic data class, and used some JavaFX Script cleverness to ensure `gridSize` and `gridDim` are always up to date whenever the data they depend upon changes. When `boxDim` is set, it begins a chain reaction which sees `gridDim`, then `gridSize`, recalculated. Strictly speaking it might have made more sense to bind

`boxDim` to the square root of `gridDim` rather than the other way around, but I don't fancy writing a square root function for a project like this.

Note, although the puzzle requires the numbers 1 to 9, we also use the number 0 (zero) to represent an empty cell. Thus the permissible values for each cell range from 0 to 9.

Our initial data class is missing a lot of important functionality, but it should be sufficient to get a user interface on screen. We can then refine and develop both the puzzle class and the interface further as the chapter progresses.

4.1.2 Our initial GUI class

So much for the puzzle grid class, what about a GUI?

Recall, JavaFX encourages software to be built in a declarative fashion, especially user interfaces. Until now this has just been a rather cute idea floating around in the ether, but right now you're about to finally see a concrete example of how this works in the real world, and why it's so powerful.

Listing 4.2 (below) is the entry point to our application, building a basic user interface for our application using the previously touted declarative syntax.

Listing 4.2 Game.fx (version 1)

```
package jfxia.chapter4;

import javafx.ext.swing.SwingButton;
import javafx.scene.Scene;
import javafx.scene.text.Font;
import javafx.stage.Stage;

def cellSize:Number = 40;                                          A

def puz = PuzzleGrid { boxDim:3 };                                 B

def gridFont = Font                                                C
{   name: "Arial Bold"                                             C
    size: 20                                                       C
};                                                                 C

Stage
{   title: "Game"
    visible: true
    scene: Scene
    {   content: for(idx in [0..<puz.gridSize])
        {   var x:Integer = (idx mod puz.gridDim);                 D
            var y:Integer = (idx / puz.gridDim);                   D
            SwingButton                                            E
            {   translateX: x * cellSize;                          E
                translateY: y * cellSize;                          E
                width: cellSize;                                   E
                height: cellSize;                                  E
                                                          E
                text: bind if(puz.grid[idx]>=1)                    E
                    "{puz.grid[idx]}" else "  "                    E
                font: gridFont                                     E
```

```
                action: function():Void                                    E
                {   var v = puz.grid[idx];                                 E
                    v = (v+1) mod (puz.gridDim+1);                         E
                    puz.grid[idx] = v;                                     E
                }                                                          E
            }                                                              E
        }
        width: puz.gridDim * cellSize
        height: puz.gridDim * cellSize
    }
};
```

A Cell dimensions
B Our puzzle class
C A font we wish to re-use
D Index to grid x/y
E Each grid cell is a Swing button

You can see from the `import` statements at the head of the source file it's pulling in several GUI based classes from the standard JavaFX APIs. The Swing wrapper classes live in a JavaFX package named, conveniently, `javafx.ext.swing`. We could use the classes in the original Swing packages directly (like other Java API classes, they *are* available) but the JFX wrappers make it easier to use common Swing components declaratively.

The `cellSize` variable defines how big, in pixels, each square in the grid will be. Our game needs a `PuzzleGrid` object, and we create one with the variable `puz`, setting `boxDim` to declaratively describe its size. After `puz` we create a font for our GUI. As we're using the same font for each grid cell, we may as well create one font object and re-use it. Again, this is done declaratively using an object literal. The final chunk of the listing, and quite a hefty chunk it is too, consists of the actual user interface code itself.

We've using Swing buttons to represent each cell in the grid, so we can easily display a label and respond to a mouse click, but let's strip the button detail away for the moment and concentrate on the outer detail of the window.

```
Stage
{   title: "Game"
    visible: true
    scene: Scene
    {   content: /*** STRIPPED, FOR NOW ***/
        width: puz.gridDim * cellSize
        height: puz.gridDim * cellSize
    }
};
```

Here's an abridged reproduction of the code we saw in listing 4.2 earlier. We see two objects being created, one nested inside the other. At the outermost level we have a `Stage`, and inside that a `Scene`. There are further objects inside the `Scene`, but the listing doesn't show them.

The `Stage` represents the link to the outside world. As this is a desktop application, the `Stage` will take the form of a window. In a web browser applet, however, the `Stage` would be an applet container. Using a common top-level object like this aids portability between profiles (desktop/web/mobile/TV).

Three variables are set on `Stage`: the window's title (as shown in the window's drag bar), the window's visibility (without this being `true` the window will not show and the application will immediately terminate) and the the window's content. The content is a `Scene` object, used to describe the root of a scene graph. We'll look at the scene graph in far more detail next chapter–for now all you need to know is it's where our UI will live.

The `Scene` object has its own variables, aside from `content`. They are `width` and `height`, and they determine the size of the grid display. The window will be sized around these dimensions, with the total window size being the `Scene` dimensions plus any borders and title (drag) bars the operating system adds to decorate the window.

There's a chunk of code missing from the middle of the snippet above, and now it's time to see what it does.

4.1.3 Building the buttons

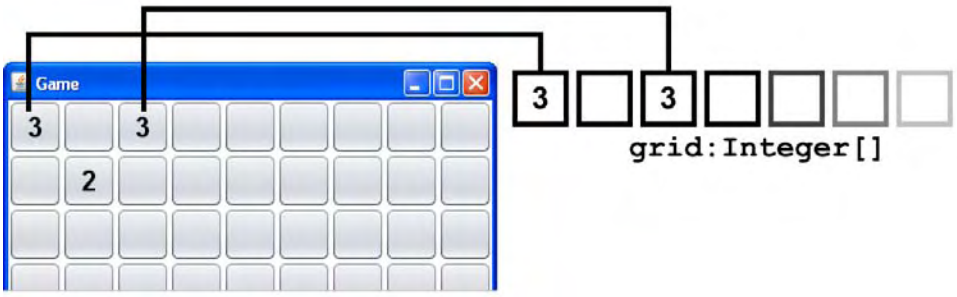
Here's a reminder of the mysterious piece of code we left out of our discussion last section:

```
content: for(idx in [0..<puz.gridSize])
{   var x:Integer = (idx mod puz.gridDim);
    var y:Integer = (idx / puz.gridDim);
    SwingButton
    {   translateX: x * cellSize;
        translateY: y * cellSize;
        width: cellSize;
        height: cellSize;

        text: bind if(puz.grid[idx]>=1)
            "{puz.grid[idx]}" else "  "
        font: gridFont
        action: function():Void
        {   var v = puz.grid[idx];
            v = (v+1) mod (puz.gridDim+1);
            puz.grid[idx] = v;
        }
    }
}
```

The code goes inside a `Scene`, which in turn provides the contents for our `Stage`, you'll recall–but what does it do? Put simply, it creates a square grid of `SwingButton` objects, tied to data inside the `puz` object. You can see the effect in figure 4.3.

Figure 4.3 The game as it appears after clicking on a few cells (note the highlight on the lower “3”). Depending upon your JRE version you'll get Ocean (left) or Nimbus (right) themed buttons.

The `for` loop runs for as many times as the puzzle's grid size, creating a fresh button for each cell. Before the button is actually created we convert the loop index, stored in `idx`, into an actual grid `x` and `y` position. Dividing by the grid width gives us the `y` position (ignoring any fractional part), while the remainder of the same division gives us the `x` position.



Each button has seven variables declaratively set. The first four are the button's position and size inside the scene, in pixels. They lay the buttons out as a grid, relying on the `cellSize` variable we created at the head of the file. The three other variables are the button's text, its font, and an event handler which fires when the button is clicked.

The button's text is bound to the corresponding element in the puzzle's grid sequence. So the first button will be bound to the first sequence element, the second to the second, and so on, as shown in figure 4.4. However, we do not want the value zero to be shown to the user, so we've constructed the bind such that it displays nothing (or rather, some space characters, so Swing won't size the button label to zero width) whenever its element is zero.

Figure 4.4 The text of each button is bound to the corresponding value in the puzzle's grid sequence.

You may recognize the `action` as an anonymous function. The function reads its corresponding element in the puzzle grid sequence, increments it, ensuring the value wraps around to zero if it exceeds the maximum number allowed, then stores it back in the sequence. Here's the clever part: because the `text` variable is bound to the sequence

element, whenever `action`'s function changes the element value, the buttons text is automatically updated.

4.1.4 Model/View/Controller, JavaFX Script style

We touched on the Model/View/Controller paradigm briefly in the introduction, and hopefully those readers familiar with the MVC concept will already have seen how this is playing out under JavaFX Script.

In languages like Java, MVC is implemented by way of interfaces and event classes. In JavaFX Script this is swept away in favor if a much cleaner approach built deep into the language syntax. The relationships between our game UI and its data are formed using binds and triggers. The *model* (game data) uses triggers to respond to input from the *view*/*controller* (UI display). In turn, the *view* binds against variables in the *model*, establishing its relationship with them as expressions.

This is how MVC works in JavaFX Script. Any code which is dependent on a model expresses that dependency using a bind, and JavaFX Script will then automatically honor that relationship without the programmer having to manually maintain it themselves. The beauty of this approach is it strips away the boilerplate classes and interfaces of other languages, like Java, distilling everything down to its purest form. If a given part of our UI is dependent on some external data, that dependency is expressed immediately (meaning "in-line") as part of its definition. It is not scattered throughout our code in disparate event handlers, interfaces and event objects, like in Java.

If there is a two way relationship between the UI and its data, a bidirectional bind (the `with inverse` syntax) can be used. For example, a text field may display the contents of a given variable in a model. If the variable changes, the text field should update automatically; if the text field is edited, the variable should update automatically. Providing the relationship is elemental in nature, in other words a direct one-to-one, a bidirectional bind will achieve this.

When I told you binds were really useful things, I wasn't kidding!

4.1.5 Running version 1

It may not be the most impressive game so far, but it gives us a solid foundation to work from. When version one of the puzzle is run it displays the puzzle grid (see figure 4.3) and responds to mouse clicks by cycling through the available numbers.

This is just a start, but already we've seen some of the power tools we learned about in the previous two chapters making a big contribution: the declarative syntax, bound variables and anonymous functions are all in full effect.

So let's continue to build up the functionality of our game by making it more useful.

4.2 Better informed and better looking: puzzle, version 2

So far we've got a basic user interface up and running, but it lacks the functionality to make it a playable game. There are two problems we need to tackle next:

- The buttons don't look particularly appealing, and it's hard to see where the boxes are on the puzzle grid.
- The game doesn't warn us when we duplicate numbers in a given group. If I, as the player, put two 3's on the same row, for example, the game does not flag this as an error.

In this section we'll remedy these faults. The first is entirely the domain of the user interface class, `Game`, while the second is predominantly the domain of the data class, `PuzzleGrid`.

4.2.1 Making the puzzle class clever, using triggers and function types

The data class was tiny in version one, with only a handful of variables to its name. To make the class more aware of the rules of the puzzle we need to add a whole host of code. Let's start with `PuzzleGrid.fx`, listing 4.3 below, and see what changes need to be made.

Listing 4.3 PuzzleGrid.fx (version 2)

```
package jfxia.chapter4;

package class PuzzleGrid
{   public-init var boxDim:Integer = 3;
    public-read def gridDim:Integer = bind boxDim*boxDim;
    public-read def gridSize:Integer = bind gridDim*gridDim;

    package var grid:Integer[] =
        for(a in [0..<gridSize]) { 0 }
        on replace current[lo..hi] = replacement                    A
        {   update();                                               A
        }                                                           A

    package var clashes:Boolean[] =                                 B
        for(a in [0..<gridSize]) false;                             B

    function update() : Void                                        C
    {   clashes = for(a in [0..<gridSize]) false;                   C
        for(grp in [0..<gridDim])                                   C
        {   checkGroup(grp,row2Idx);                                C
            checkGroup(grp,column2Idx);                             C
            checkGroup(grp,box2Idx);                                C
        }                                                           C
    }                                                               C

    function checkGroup                                             D
    (   group:Integer ,                                             D
        func:function(:Integer,:Integer):Integer                    D
    ) : Void                                                        D
    {   var freq = for(a in [0..gridDim]) 0;                        D
        for(pos in [0..<gridDim])                                   D
        {   var val = grid[ func(group,pos) ];                      D
            if(val > 0) { freq[val]++; }                            D
        }                                                           D
        for(pos in [0..<gridDim])                                   D
        {   var idx = func(group,pos);                              D
```

```
            var val = grid[idx];                                        D
            clashes[idx] = clashes[idx] or (freq[val]>1);               D
        }                                                               D
    }                                                                   D

    // Horizontal groups
    function row2Idx(group:Integer,pos:Integer) : Integer
    {   return group*gridDim + pos;
    }
    // Vertical groups
    function column2Idx(group:Integer,pos:Integer) : Integer
    {   return group + pos*gridDim;
    }
    // Box groups
    function box2Idx(group:Integer,pos:Integer) : Integer
    {   var xOff = (group mod boxDim) * boxDim;
        var yOff = ((group/boxDim) as Integer) * boxDim;
        var x = pos mod boxDim;
        var y = (pos/boxDim) as Integer;
        return (xOff+x) + (yOff+y)*gridDim;
    }
}
```

A Trigger an update to 'clashes'
B Does this cell clash with another?
C Update clashes sequence, above
D Check a given group for clashes

Whew! That's quite bit of code to be added in one go, but don't panic, it's all rather straight forward when you know what it's meant to do.

The purpose of listing 4.3 is to update a new sequence, called `clashes`, which hold flags set to `true` if a given cell currently conflicts with others, and `false` if it does not. The user interface can then bind to this sequence, changing the way a grid cell is displayed to warn the player of any duplicates.

The function `update()` clears the `clashes` sequence, then checks each group in turn. In our basic 9 x 9 puzzle there are twenty-seven groups: nine rows, nine columns and nine boxes. Each group has nine positions it needs checking for duplicates. However, most of the work is deferred to the function `checkGroup()`, which handles the checking of an individual group. Let's take a closer look at this function, so we can understand how it fits into `update()`.

4.2.2 Group checking up close: function types

Here's the `checkGroup()` function we're looking at, reproduced on its own to refresh your memory and save ambiguity:

```
function checkGroup
(   group:Integer ,
    func:function(:Integer,:Integer):Integer
) : Void
{   var freq = for(a in [0..gridDim]) 0;
    for(pos in [0..<gridDim])
    {   var val = grid[ func(group,pos) ];
        if(val > 0) { freq[val]++; }
```

```
        }
        for(pos in [0..<gridDim])
        {   var idx = func(group,pos);
            var val = grid[idx];
            clashes[idx] = clashes[idx] or (freq[val]>1);
        }
    }
```

The first four lines are the function's signature; unfortunately it's quite long, so I split it over four separate lines in an attempt to make it more readable. We can see the function is called `checkGroup`, and accepts two parameters: an Integer and a function type. The function type accepts two `Integer` variables and gives a single `Integer` in return. The `Void` on the end signifies `checkGroup()` has no return value.

So, you're probably wondering, why do we need to pass a function to `checkGroup()`? Well, think about it: when we're checking each position in a row group we're working horizontally across the grid, when we're checking each position in a column group we're working vertically down the grid, and for a box group we're working line by line within a portion of the grid. Figure 4.5 demonstrates this, below.

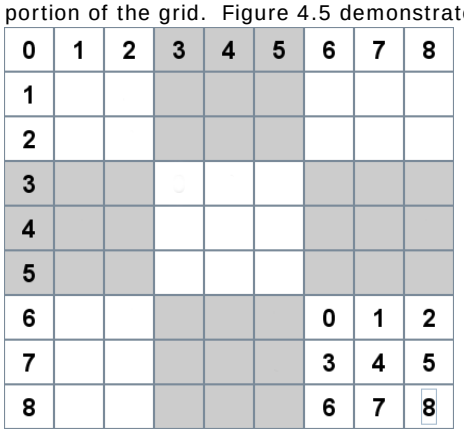


Figure 4.5 Co-ordinate translations for column, row and box groups.

We have three different ways of translating group and position to grid co-ordinates:

- For rows the group is the Y co-ordinate in the grid and the position is X co-ordinate. So position 0 of group 0 will be (0,0) on the grid, and position 2 of group 1 will be (2,1) on the grid.
- For columns the group is the X co-ordinate in the grid and the position is the Y co-ordinate. So position 2 of group 1 will be (1,2) when translated into grid co-ordinates.
- For boxes we need to do some clever math to translate the group to a sub-section of the grid, so group 8 (in the southeast corner) will have its first cell at co-ordinates (6,6). We then need to do some more clever math to turn the position into an offset within this sub-grid. Position 1, for example, would be offset (1,0), giving us an absolute position of (7,6) on the grid when combined with group 8.

Now, the code which actually checks for duplicates within a given group is identical, as we've just seen—all that differs is the way the group type translates its group number and position into grid co-ordinates. So the function passed in to `checkGroup()` abstracts away this translation. The two values it accepts are the group and the position. The value it returns is the grid co-ordinate, or rather the index in the grid sequence which corresponds with said group and position. (One of the benefits of storing the grid as a single dimension sequence is that we don't need to figure out a way to return two values, an 'x' and a 'y', from these three translation functions.)

Now that we understand what the passed in function does, let's quickly examine the code inside `checkGroup()` to see what it does. It's broken up into two stages:

```
var freq = for(a in [0..gridDim]) 0;
for(pos in [0..<gridDim])
{   var val = grid[ func(group,pos) ];
    if(val > 0) { freq[val]++; }
}
```

Here's the first stage reproduced on its own. We kick off by defining a new sequence called `freq`, with enough space for each unique number in our puzzle. This will hold the frequency of the numbers in our group. Recall, we're using zero to represent an empty cell, so to make the code easier we've allowed space in the sequence for zeros. Then we extract each position in the group, using the parameter function to translate group/position to a `grid` index. We increment the frequency corresponding to the value at said grid index—so if the value was 1 then `freq[1]` would get incremented.

This builds us a table of how often each value occurs in the group. Now we want to act on that data.

```
for(pos in [0..<gridDim])
{   var idx = func(group,pos);
    var val = grid[idx];
    clashes[idx] = clashes[idx] or (freq[val]>1);
}
```

The second stage of the `checkGroup()` function is reproduced above. It should be quite obvious what needs doing: we take a second pass over each position in the group, pulling out its value once again with help from our translation function, then setting the flag in `clashes` if said value appears in the group more than once (signifying a clash!)

Note: a given cell in the puzzle grid may cause a clash in some groups, but not others. It is important, therefore, that we preserve the clashes already discovered with other groups. This is why the clash check is or'd with the current value in the `clashes` sequence, rather than simply overwriting it.

4.2.3 Firing the update: triggers

Now we know how each group is checked, and we've seen the power of using function types to allow us to re-use code with 'plug-in-able' variations, but we still need to complete the picture. The function `update()` will call `checkGroup()` twenty-seven times (assuming a standard 9x9 grid), but what makes `update()` run?

Perhaps a better question might be “when should `update()` run?”, to which the answer should surely be “whenever the `grid` sequence is changed!”

We *could* wire something into our button event handler to always call `update()`, but this would be exposing the `PuzzleGrid` class's inner mechanics to another class, something which we should avoid if we can. But why not wire something into the actual `grid` sequence itself?

```
package var grid:Integer[] =
    for(a in [0..<gridSize]) { 0 }
        on replace current[lo..hi] = replacement
        {   update();
        }
```

Using a trigger we can ensure `update()` runs whenever the `grid` sequence is modified. It's a simple, effective, and clean solution which ensures the `clashes` sequence will never get out of step with `grid` .

4.2.4 Better looking GUI: playing with the underlying Swing component

We've managed to *supe up* the puzzle class itself by making it responsive to duplicates under the rules of the puzzle. We can now exploit that functionality in our GUI, but we also need to make the game look more appealing.

Listing 4.4 gives us version two of the `Game` class:

Listing 4.4 Game.fx (version 2)

```
package jfxia.chapter4;

import javax.swing.JButton;
import javax.swing.border.LineBorder;
import javafx.ext.swing.SwingButton;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.stage.Stage;


def gridCol1 = Color.WHITE;                                          A
def gridCol2 = Color.web("#CCCCCC");                                 A
def lineCol = Color.GRAY;                                            B
def border = new LineBorder(lineCol.getAWTColor());                  B

def cellSize:Number = 40;

def puz = PuzzleGrid { boxDim:3 };

def gridFont = Font
{   name: "Arial Bold"
    size: 20
};

Stage
{   title: "Game"
    visible: true
    scene: Scene
```

```
    {   content: for(idx in [0..<puz.gridSize])
        {   var x:Integer = (idx mod puz.gridDim);
            var y:Integer = (idx / puz.gridDim);
            var b = SwingButton                                          C
            {   translateX: x * cellSize;
                translateY: y * cellSize;
                width: cellSize;
                height: cellSize;

                text: bind if(puz.grid[idx]>=1)
                    "{puz.grid[idx]}" else "  "
                font: gridFont
                foreground: bind                                         D
                    if(puz.clashes[idx]) Color.RED                       D
                    else Color.GRAY                                      D
                action: function():Void
                {   var v = puz.grid[idx];
                    v = (v+1) mod (puz.gridDim+1);
                    puz.grid[idx] = v;
                }
            };

            x/=puz.boxDim;  y/=puz.boxDim;                               E
            var bg = if((x mod 2)==(y mod 2)) gridCol1                   E
                else gridCol2;                                           E

            var jb = b.getJComponent() as Jbutton;                       F
            jb.setContentAreaFilled(false);                              F
            jb.setBackground(bg.getAWTColor());                          F
            jb.setOpaque(true);                                          F
            jb.setBorder(border);                                        F
            jb.setBorderPainted(true);                                   F
            b;
        }
        width: puz.gridDim * cellSize
        height:puz.gridDim * cellSize
    }
};
```

A Two tone background
B Lines between grid cells
C A Button reference
D Bind the foreground color to clashes
E Background color from index
F Manipulate the button via Swing

The above cyan [CA]'s are code continuations, for wrapped lines.

Listing 4.4 adds a couple of new imports at the head of the file, and two new variables: `gridCol1` and `gridCol2`. These will help us to change the background color of the buttons, to represent the boxes on the grid. We'll use the `border` variable to give us a gray pin line around each box, as seen below in figure 4.6.

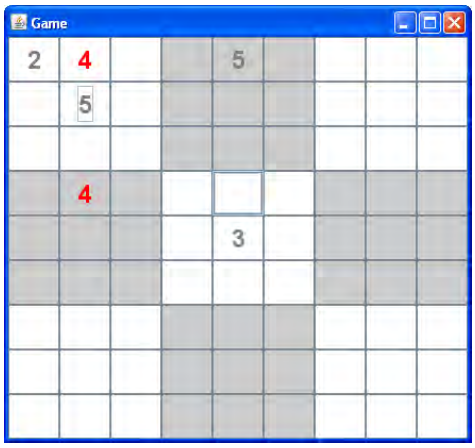


Figure 4.6 The restyled user interface, with differentiated boxes using background color, and duplicate warnings using foreground (text) color.

All the changes center around the button sequence being created and added into the `Scene`. You'll note from figure 4.6 the button's `foreground` color is now bound to the `clashes` sequence which we developed previously. You'll recall, when the contents of `puz.grid` change, `puz.clashes` is automatically updated via the trigger we added to the `PuzzleGrid` class. With this bind in place the user interface's buttons are immediately recolored to reflect any change in the clash status.

But those aren't the only changes we've made. In version one (listing 4.2) we just added a new `SwingButton` straight into the scene graph, but now we capture it in a variable reference to manipulate it further, before it gets added. After the button is created we extract the underlying Swing component using the function `getJComponent()`, giving us full access to all its methods. First we remove the shaded fill effect. Next we set our own flat color background, creating a checkerboard effect for the boxes, and ensure the background is painted by making the button opaque. Finally we assign the `border` we created earlier, to create a gray pin line around each button.

Note that the background color is calculated by translating the cell's raw grid coordinate into a box coordinate (just scale them by the box size) and using this to assign colors based on a checkerboard pattern.

At the end of the loop we restate the variable used to hold the button reference. Because JavaFX Script is an expression language, this becomes our returned value to the loop's sequence.

Boxing clever: how to create a checkerboard pattern

In the `box2Idx()` function we witnessed in `PuzzleGrid`, and now with the checkerboard background pattern, we've had to do some clever math to figure out where

the boxes are. Perhaps you're not interested in how this was done—but for those who are curious, here's an explanation...

We need to convert a sequence index (from 0 to 80, assuming a standard 9x9 grid) to a box co-ordinate (0,0) through (2,2) assuming nine boxes. The pattern itself is quite easy to produce once you have these co-ordinates: if x and y are both odd or both even we use one color; if x and y are odd/even or even/odd, we use the other color. We can figure out whether an number is odd or even by taking the remainder (the `mod`) of a division by two: odd numbers result in 1, even numbers result in 0. Try it for yourself on a piece of paper if you don't believe me.

So, let's assume we're given a `grid` sequence index, like 23, should this cell be shaded with a white or a gray background? First we convert the number 23 into a co-ordinate on the grid. The y co-ordinate is the number of times the grid dimension will divide into the our number: 23 divided by 9 is 3. The x co-ordinate is the remainder of this division: 23 mod 9 is 2. (Remember, the co-ordinates start a zero, not one.) Therefore index 23 is grid co-ordinate (2,3). But we need to translate this into a box co-ordinate, which is easily done by scaling it by the box size: 2 divided by 3 is 0, and 3 divided by 3 is 1. So grid index 23 becomes grid co-ordinate (2,3), becomes box co-ordinate (0,1). We then compare the odd-ness and even-ness of these co-ordinates to determine which background shade to use.

What about the `box2Idx()` function? Well, its similar in principle, except we're almost working backwards: we're given a group and a position, and we need to work out the `grid` sequence index. To do this we first need to find the origin (northwest) co-ordinate of the box representing the group in the grid, which we can do my dividing and mod'ing the group by the box size to get x and y co-ordinates. We do the same thing to the position to get the co-ordinate offset within the box. Then we add the offset on to the origin, to get the absolute x and y within the grid. Finally, to convert the grid x,y to an index we multiple y by the grid dimension, and add on x.

With our new version of the game class in place, we're ready to try running the code again and seeing how the changes play out on the desktop.

4.2.5 Running version 2

So thanks to some clever trigger action, and a little bit of Swing coding, we've managed to get a puzzle game which now looks more the part, and can warn players when they enter duplicate numbers within a group. See figure 4.6 for how the game currently looks.

It may seem like we're still a million miles away from a completed game, but actually we're in the home straight, and the finishing line is within sight. So let's push on to the next, and final, version of the game.

4.3 Game on: puzzle version 3

We're almost there, the puzzle is nearly complete. But what work do we have left to do on our game? Let's make a list:

1. We need to add actual numbers for the start point of the puzzle, otherwise the grid is just empty, and the puzzle wouldn't be very... well... puzzling.
2. We need to lock these starting numbers, so the player can't accidentally change them.
3. We need to notify the player when they've solved the puzzle. It would also be nice to inform them how many empty grid cells, and how many clashes, they currently have.

This is really just a mopping up exercise, dealing with all the outstanding issues necessary to make the puzzle work. Yet there's still opportunity for learning, and practicing our JavaFX skills, as you'll shortly see.

4.3.1 Adding stats to the puzzle class

In order to inform the player of how many clashes and empty cells we have, the puzzle class needs some added functionality. We also need the class to provide some way of fixing the starting cells, so the GUI knows not to change them.

Listing 4.5 is the final version of the class, with all the new code added:

Listing 4.5 PuzzleGrid.fx (version 3)

```
package jfxia.chapter4;

package class PuzzleGrid
{   public-init var boxDim:Integer = 3;
    public-read def gridDim:Integer = bind boxDim*boxDim;
    public-read def gridSize:Integer = bind gridDim*gridDim;

    package var grid:Integer[] =
        for(a in [0..<gridSize]) { 0 }
        on replace current[lo..hi] = replacement
        {   update();
        }

    package var clashes:Boolean[] =
        for(a in [0..<gridSize]) false;

    package var fixed:Boolean[] =                                    A
        for(a in [0..<gridSize]) false;                              A
                                                                     A
    package var numEmpty = 0;                                        A
    package var numClashes = 0;                                      A
    package def completed:Boolean = bind                             A
        ((numEmpty==0) and (numClashes==0));                         A

    public function fixGrid() : Void                                 B
    {   for(idx in [0..<gridSize])                                   B
        {   fixed[idx] = (grid[idx]>0);                              B
        }                                                            B
    }                                                                B

    function update() : Void
    {   clashes = for(a in [0..<gridSize]) false;
```

```
        for(grp in [0..<gridDim])
        {   checkGroup(grp,row2Idx);
            checkGroup(grp,column2Idx);
            checkGroup(grp,box2Idx);
        }
        checkStats();                                                    C
    }

    function checkGroup
    (    group:Integer ,
        func:function(:Integer,:Integer):Integer
    ) : Void
    {   var freq = for(a in [0..gridDim]) 0;
        for(pos in [0..<gridDim])
        {    var val = grid[ func(group,pos) ];
            if(val > 0) { freq[val]++; }
        }
        for(pos in [0..<gridDim])
        {   var idx = func(group,pos);
            var val = grid[idx];
            clashes[idx] = clashes[idx] or (freq[val]>1);
        }
    }

    function checkStats() : Void                                         D
    {   numEmpty = 0;                                                    D
        numClashes = 0;                                                  D
        for(idx in [0..<gridSize])                                       D
        {   if(grid[idx]==0)  numEmpty++;                                D
            if(clashes[idx])  numClashes++;                              D
        }                                                                D
    }                                                                    D

    // Horizontal groups
    function row2Idx(group:Integer,pos:Integer) : Integer
    {   return group*gridDim + pos;
    }
    // Vertical groups
    function column2Idx(group:Integer,pos:Integer) : Integer
    {   return group + pos*gridDim;
    }
    // Box groups
    function box2Idx(group:Integer,pos:Integer) : Integer
    {   var xOff = (group mod boxDim) * boxDim;
        var yOff = ((group/boxDim) as Integer) * boxDim;
        var x = pos mod boxDim;
        var y = (pos/boxDim) as Integer;
        return (xOff+x) + (yOff+y)*gridDim;
    }
}
```

A **The new variables**
B **Fix the static starting cells**
C **Call the stats updater**
D **Count empty and clashing cells**

Four new variables have been added in listing 4.5: the first, `fixed`, is a sequence which denotes which cells in the grid should not be changeable. The next three provide basic stats about the puzzle: `numEmpty`, `numClashes`, and `completed`.

We've also added a new function, `fixGrid()`, which walks over the puzzle grid and marks any cells which are non-zero in the `fixed` sequence. The GUI class can call this function to lock all existing cells in the puzzle, but you may be wondering why we didn't use some clever device like a bind or a trigger to automatically update the `fixed` sequence?

The `grid` sequence gets updated frequently—indeed, each time the player changes the value of a cell in the puzzle. We only need the `fixed` sequence to update when the `grid` is initially loaded with the starting values of the puzzle. Now, we could rather cleverly re-write the trigger to spot when the entire `grid` is being written, rather than a single cell (it can see how many cells are being changed at once, after all), but this might cause confusion later on. For example, suppose we added a load/save feature to our game? Restoring the grid after a load operation would cause the trigger to mistakenly fix all the existing non-zero cells—including those added by the player. Some may be wrong, indeed some may be clashes! How would the player feel if they were unable to change them?

For all the power JavaFX Script gives us, it must be acknowledged sometimes the most basic solution is the best, even if it doesn't give us a chance to show fellow programmers just how clever-clever we are.

We have one final new function in our class: `checkStats()` simply populates the `numEmpty` and `numClashes` variables. It's called from the `update()` function, so it will run whenever the `grid` sequence is changed. The `completed` variable is bound to these variables, and will become true when both are zero.

And that pretty much wraps it up for the puzzle class itself. Almost there! Let's now turn to the final piece of the puzzle (groan!), the GUI.

4.3.2 Finishing off the puzzle grid GUI

If you survived the horrendous pun at the end of the last section you'll know this is the bit where we pull everything together in one final burst of activity on the GUI class, to complete our puzzle game.

We have two aims with these modifications:

1. Provide an actual starting grid, to act as a puzzle. The game is pointless without it.
2. Plug in a status line at the bottom of the grid display, to inform the player of empty cells, clashing cells, and a successfully complete puzzle.

We'll look at the former in this section, and the latter next section.

You might think these changes would be pretty mundane, but I've thrown in a layout class to keep you on your toes. Check out listing 4.6, next.

Listing 4.6 Game.fx (version 3)

```
package jfxia.chapter4;
```

```
import javax.swing.JButton;
import javax.swing.border.LineBorder;
import javafx.ext.swing.SwingButton;
import javafx.ext.swing.SwingLabel;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.stage.Stage;


def gridCol1 = Color.WHITE;
def gridCol2 = Color.web("#CCCCCC");
def lineCol = Color.GRAY;
def border = new LineBorder(lineCol.getAWTColor());

def cellSize:Number = 40;

var puz = PuzzleGrid
{   boxDim: 3
    grid:                                                                A
    [   5,3,0 , 0,7,0 , 0,0,0 ,                                          A
        6,0,0 , 1,9,5 , 0,0,0 ,                                          A
        0,9,8 , 0,0,0 , 0,6,0 ,                                          A
        8,0,0 , 0,6,0 , 0,0,3 ,                                          A
        4,0,0 , 8,0,3 , 0,0,1 ,                                          A
        7,0,0 , 0,2,0 , 0,0,6 ,                                          A
        0,6,0 , 0,0,0 , 2,8,0 ,                                          A
        0,0,0 , 4,1,9 , 0,0,5 ,                                          A
        0,0,0 , 0,8,0 , 0,7,9                                            A
    ]                                                                    A
};
puz.fixGrid();                                                           B

var gridFont = Font
{   name: "Arial Bold"
    size: 20
};

Stage
{   title: "Game"
    visible: true
    scene: Scene
    {   content:
        [   for(idx in [0..<puz.gridSize])                               C
            {   var x:Integer = (idx mod puz.gridDim);
                var y:Integer = (idx / puz.gridDim);
                var b = SwingButton
                {   translateX: x * cellSize;
                    translateY: y * cellSize;
                    width: cellSize;
                    height: cellSize;

                    text: bind if(puz.grid[idx]>=1)
                        "{puz.grid[idx]}" else "  "
                    font: gridFont
                    foreground: bind
```

```
                        if(puz.clashes[idx]) Color.RED
                        else if(puz.fixed[idx])  Color.BLACK
                        else Color.GRAY
                    action: function():Void
                    {   if(puz.fixed[idx]) { return; }
                        var v = puz.grid[idx];
                        v = (v+1) mod (puz.gridDim+1);
                        puz.grid[idx] = v;
                    }
                };

                x/=puz.boxDim;  y/=puz.boxDim;
                var bg = if((x mod 2)==(y mod 2)) gridCol1
                    else gridCol2;

                var jb = b.getJComponent() as JButton;
                jb.setContentAreaFilled(false);
                jb.setBackground(bg.getAWTColor());
                jb.setOpaque(true);
                jb.setBorder(border);
                jb.setBorderPainted(true);
                b;
            } ,
            HBox                                                          D
            {   translateY: bind puz.gridDim * cellSize                   D
                content:                                                  D
                [   SwingLabel                                            D
                    {   text: bind "  Empty: {puz.numEmpty}"              D
                            "   Clashes: {puz.numClashes}"                D
                    } ,                                                   D
                    SwingLabel                                            D
                    {   text: bind if(puz.completed)                      D
                            "Complete!" else " "                          D
                        foreground: Color.GREEN                           D
                        translateX: 50                                    D
                        width: 200                                        D
                    }                                                     D
                ]                                                         D
            }                                                             D
        ]
        width: puz.gridDim * cellSize
        height:puz.gridDim * cellSize + 20                                E

    }
};
```

A Some puzzle data, at last!
B Fix the initial puzzle cells
C Grid loop now inside larger sequence
D The status panel
E Allow for status line

We can see a couple of new imports at the head of the file—one is a Swing label class, and the other is the promised JFX layout class.

You'll note the `grid` variable of the `PuzzleGrid` class is now being set, and `fixGrid()` is being called to lock the initial puzzle numbers in place. You could provide your own puzzle

data here if you want—I used the data which illustrates the Wikipedia Sudoku article as an example.

In this version of the game we'll be adding more elements to the scene graph beyond those created by the `for` loop. For this reason the loop (which creates a sequence of Swing buttons) has been moved inside a set of square brackets, effectively wrapping it inside a larger sequence. Embedded sequences like this are expanded in place, you'll recall, so the buttons yielded from the loop are expanded into the outer sequence, and our new elements (see later) are added after them.

Taking a closer look at the `Button` definition we see a couple of minor, but important, additions. Here's the snippet of code we're taking about, extracted out of the main body of the button creation:

```
foreground: bind
    if(puz.clashes[idx]) Color.RED
    else if(puz.fixed[idx])  Color.BLACK
    else Color.GRAY
action: function():Void
{   if(puz.fixed[idx]) { return; }
    var v = puz.grid[idx];
    v = (v+1) mod (puz.gridDim+1);
    puz.grid[idx] = v;
}
```

In the `foreground` code we now look for and colorize fixed cells as solid black. And to compliment this, in the event handler, we now check for unchangeable cells, exiting if one is clicked without modifying its contents. These two minor changes, coupled to the work we did with the `PuzzleGrid` class, ensure the starting numbers of a puzzle will not be editable.

4.3.3 Adding a status line to our GUI with Swing's BorderPanel and Label

The bulk of the changes come with the introduction of a status line to the foot of the GUI declaration. Here, for your convenience, is the code once more, devoid of its surrounding clutter.

```
HBox
{   translateY: bind puz.gridDim * cellSize
    content:
    [   SwingLabel
        {   text: bind "  Empty: {puz.numEmpty}"
                "   Clashes: {puz.numClashes}"
        } ,
        SwingLabel
        {   text: bind if(puz.completed)
                "Complete!" else " "
            foreground: Color.GREEN
            translateX: 50
            width: 200
        }
    ]
}
```

This code is placed inside a larger sequence, used to populate the `Scene`. The UI elements it defines appears after all the grid buttons, which were created using a for loop as we saw earlier.

You'll immediately notice an HBox, which is JavaFX's horizontal layout panel. It controls how its children are positioned on screen. Look at figure 4.7 and you'll see another example of HBox in action. The gray shapes are arranged, from the left to the right, in the order they appear inside the `content` sequence.

Figure 4.7 Elements inside an HBox are laid out side by side, from left to right.

Like the Scene class, HBox accepts a sequence for its contents, but doesn't add any new graphics itself. It merely positions its children on the display. It has a sibling, VBox, which does the same thing, but vertically. Using these layout *panels* we can place screen objects in relation to one another, without resorting to absolute x/y positioning as we did with the button grid.

Layout control

JavaFX release 1.0 (December 2008) has little in the way of layout panels. The Swing based panels from in the preview releases were removed, but not replaced with alternatives. Hopefully future editions will have more layout classes.

To ensure the status line is at the foot of the display, we use its `translateY` variable to lower it below the button grid. It you check out the Scene you'll note it has had its `height` amended to accommodate the new content. You can see how it looks in figure 4.8.

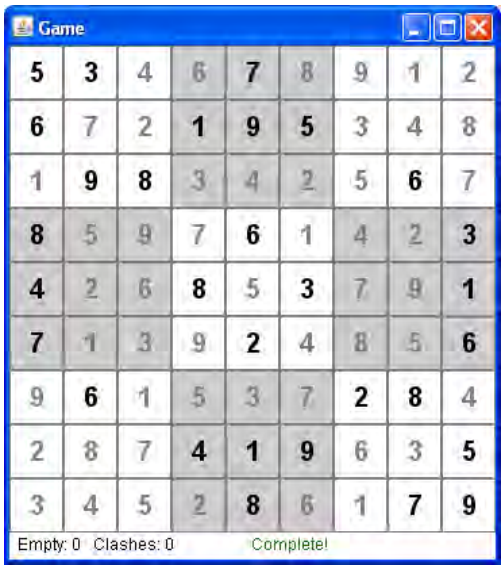


Figure 4.8 The puzzle game with its status panel, implemented using HBox.

To implement the status bar we need to arrange five pieces of text, so we'll use SwingLabel classes. These are the equivalent of the Swing JLabel class, designed to show small quantities of text (typically used for labeling, hence the name) on our user interface. We can group the "Empty" and "Clashes" text into one label, but the "Completed!" text needs to be a different color. So we need two labels.

```
SwingLabel
{   text: bind "  Empty: {puz.numEmpty}"
        "   Clashes: {puz.numClashes}"
} ,
SwingLabel
{   text: bind if(puz.completed)
        "Complete!" else " "
    foreground: Color.GREEN
    translateX: 50
    width: 200
}
```

Here's the declarative code for the two labels again, and you can immediately see how both have their text bound to the variables in the puz object. The first merely uses an embedded string expression to display data from the puzzle object, while the second uses a bound if expression which either shows the "Complete!" message, or just a space character.

The second label has been pushed over by 50 pixels, to maintain a gap between itself and the first label. It has been set to 200 pixels wide, to ensure it will be big enough to accommodate the "Complete!" message once the player solves the puzzle.

So there we go—our GUI! Whew!

4.3.4 Running version 3

We now have a functioning number puzzle game, as depicted in figure 4.8. Clicking on a changeable cell causes its numbers to cycle through all the possibilities. Duplicate numbers (clashes) are shown in red, while the fixed numbers of the initial puzzle are shown in black. The status bar keeps track of our progress, and informs us when we have a winning solution.

Almost all of this was done by developing an intelligent puzzle class, `PuzzleGrid`, which automatically responds to changes in the grid with updates to its other data. The GUI in turn is bound to this data, using it to colorize cells in the grid and show status information.

A single click on a button sets off a chain reaction, updating the grid, which updates the other status variables, which updates the GUI. Once the `bind` and `trigger` relationships are defined the code runs automatically, without us having to prompt it each time the grid is altered. In our example the grid only gets altered from one place (the anonymous function event handler on each button) so this might not seem like much of a saving, but imagine how much easier life will be if we expanded our game to include a load/save feature, or a hint feature—both of which alter the grid contents. Indeed, it becomes no extra work at all, so long as the relationships between each variable are well defined through binds.

4.4 Other Swing components

In our lightning tour of JavaFX's Swing support we looked at a couple of common widgets , namely `SwingButton` and `SwingLabel`. We also looked at some core scene graph containers, like `Stage`, `Scene` and `HBox`. Obviously it's hard to create an example project which would include every different type of widget, so here's a quick rundown of just a few key UI classes we didn't look at:

- `SwingCheckBox`—a button which is either checked or unchecked.
- `SwingComboBox`—displays a drop down list of items, optionally with a free text box.
- `SwingList` and `SwingListItem`—displays a list of items from which the user can select.
- `SwingRadioButton`, `SwingToggleGroup`—together these classes allow for single selectable groups of buttons, effectively toggle buttons in which only one button from a given group can be selected.
- `SwingScrollPane`—allows large UI content to be displayed through a restricted viewport, with scrollbars for navigation. Useful if you have a big panel of widgets, for example, which you want to display inside a scrollable area.
- `SwingSlider`—a thumb and track widget, for selecting a value from a range of possibilities using a mouse.
- `SwingTextField`—provides text entry facilities, unsurprisingly.

These are just a few of the classes in the `javafx.ext.swing` package, and Swing itself provides many more. You could get some practice with them by expanding our puzzle application—for example, how about a toggle which switches the highlighting of clashing cells on or off? This could be done by way of a `SwingCheckBox`, perhaps?

4.5 Using bind to validate forms

This chapter has been a fun way to introduce key JavaFX Script language constructs, like binds and triggers. These tools are very useful, particularly for things like form validation. Before we move on therefore, let's take a detour to look at an example, by way of listing 4.7 below.

Listing 4.7 Using bind for form validation

```
import javafx.ext.swing.*;
import javafx.scene.Scene;
import javafx.scene.shape.Circle;
import javafx.scene.input.KeyEvent;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

var ageTF:SwingTextField;
def ageValid:Boolean = bind checkRange(ageTF.text,18,65);

Stage
{   scene: Scene
    {   content: VBox
        {   translateX: 5;  translateY: 5;
            content:
            [   HBox
                {   content:
                    [   SwingLabel { text: "Age: "; },
                        ageTF = SwingTextField { columns: 10 },
                        Circle
                        {   def sz:Number = bind
                                ageTF.layoutBounds.height/2;
                            translateX: 5;
                            centerX: bind sz;
                            centerY: bind sz;
                            radius: bind sz/2;
                            fill: bind if(ageValid) Color.LIGHTGREEN
                                else Color.RED;
                        }
                    ]
                }
                SwingButton
                {   text: "Send";
                    enabled: bind ageValid;
                }
            ]
        }
        width: 190;  height: 65;
    }
}
```

```
function checkRange(s:String,lo:Integer,hi:Integer) :Boolean
{   try
    {   def i:Integer = Integer.parseInt(ageTF.text);
        return (i>=lo and i<=hi);
    }
    catch(any) { return false; }
}
```

This self-contained demo uses a function, `checkRange()`, to validate the contents of a text field. Depending upon the validity state, an indicator circle changes and the "Send" button switches between disabled and enabled. We'll be dealing with raw shapes like circle next chapter, so don't worry too much about the unfamiliar code right now; the important part is in the binds involving `ageValid`.

The circle starts red, and the button disabled. As we type, these elements update, as shown in figure 4.9. An age between 18 and 65 changes the circle color and enables the button, all thanks to the power of binds (you may need to squint to see the Swing button's subtle appearance change). The `ageValid` variable is bound to a function for checking whether the text field content is an integer, within the specified range. This variable is in turn bound by the circle and the "Send" button.

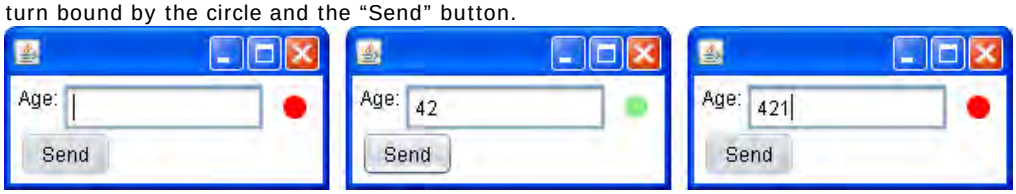


Figure 4.9 Age must be between 18 and 65 inclusive. Incorrect content shows a red circle and disables "Send" (left and right); correct content shows a light green circle and enables "Send" (middle).

In a real application we would have numerous form fields, each with their own validity boolean. We would use all these values to control the "Send" button's behavior. We might also develop a convenience class using the circle, pointing it at a UI component (for sizing) and binding it to the corresponding validity boolean. In the next chapter we'll touch on creating custom graphic classes like this, but for now just study the way bind is used to create automatic relationships between parts of our UI.

4.6 Summary

In this chapter we developed a working number puzzle game, complete with fully responsive desktop GUI. And in less than 200 lines of code... not bad!

Assuming you're still reading and haven't fallen foul to our fiendish number puzzle game (can I remind you, the screen shots give the solution away, so there's really no excuse!), you've hopefully learned a lot about writing JavaFX code during the course of this chapter. Although the number puzzle wasn't the most glamorous of applications in terms of visuals, it was still fun (I hope!), and afforded us some much needed practice with the JFX language. And that's all we need, right now.

Sure, the game could be improved upon, for example it would be nice for the cells to respond to keyboard input so the player didn't have to cycle through each number in turn, but I'll leave that as an exercise to the reader. Hopefully you've seen enough of JavaFX by now, you can extract the required answers from the API documentation and implement them yourself.

Next chapter we're getting close up and personal with the scene graph, JavaFX's backbone for presenting and animating flashy visuals. So make sure you pack your ultra-trendy shades...

5

Behind the scene graph

In the last chapter, we looked at building a rather traditional user interface with Swing. Although Swing is an important Java toolkit for interface development, it isn't central to JavaFX's way of working with graphics. JavaFX comes at graphics programming from a very different angle, with a focus more on free-form animation, movement and effects, contrasting to Swing's rather rigid widget controls. In this chapter we'll be taking our first look at how JFX does things, constructing a solid foundation onto which we can build in future chapters with ever more sophisticated and elaborate graphical effects.

The project we'll be working on is more fun than practical. The idea is to create something visually interesting with comparatively few lines of source code. Certainly far less than we'd expect if we were forced to build the same application using a language like Java or C++. One of the driving factors behind JFX is to allow rapid prototyping and construction of computer visuals and effects, and it's this speed and ease of development I hope to demonstrate as we progress through the chapter.

We'll be exploring what's known as the *scene graph*, the very heart of JavaFX's graphics functionality. We touched on the scene graph briefly last chapter, now it's time to get better acquainted with it. The scene graph is a remarkably different beast to the Java2D library typically used to write Java graphics, but it is important to remember one is not a replacement for the other. Both JavaFX's scene graph and Java2D provide different means of getting pixels on screen, and each has its strengths and a weaknesses. For slick, colorful, visuals the scene graph model has many advantages over the Java2D model—we'll be seeing exactly why that is in the next section.

5.1 What is a scene graph?

There are two ways of looking at graphics: a blunt low level “throw the pixels on screen” approach, and a higher level abstraction which sees the display as constructed from recognizable primitives, like lines, rectangles and bitmap images.

The first is what's called an *immediate mode* approach, and the second a *retained mode* approach. In immediate mode each element on the display is instructed to draw itself into a designated part of the display immediately—no record is kept of what is being drawn (other than the destination bitmap itself, of course.) By comparison, in retained mode a tree structure is created detailing the type of graphic elements resident on the display—the upkeep of this (rendering it to screen) is no longer the responsibility of each element.

Figure 5.1 is a representation of how the two systems work.

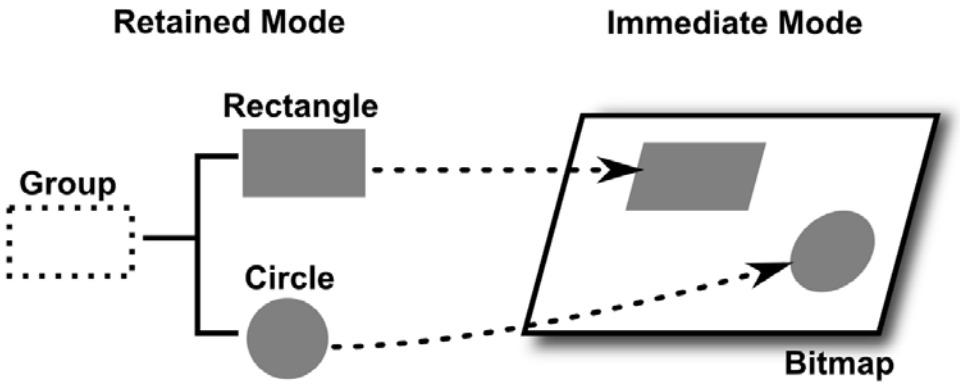


Figure 5.1 A symbolic representation of retained mode and immediate mode. The former sees the world as a hierarchy of graphical elements, the latter as just pixels.

We can characterize the immediate mode approach like so: "when it's time to redraw the screen I'll point you towards your bit of it, and you take charge of drawing what's necessary". Meanwhile the retained mode approach could be characterized as, "Tell me what you look like, and how you fit in with the other elements on the display, and I'll make sure you're always drawn properly, at the correct position, and at the right time".

This offloading of responsibility allows any code using the retained mode model to concentrate on other things, like animating and otherwise manipulating its elements, safe in the knowledge all changes will be correctly reflected on screen.

So, what is a scene graph? The *scene graph* is, quite simply, the structure of display elements to be maintained on screen in a retained mode system.

5.1.1 Nodes: the building blocks of the scene graph

The elements of the scene graph are known as nodes. Some nodes describe drawing primitives, like a rectangle, a circle, a bitmap image or a video clip. Other nodes act as grouping devices; like directories in a file system, they enable other nodes to be collected together, and a tree like structure to be created. This tree like structure is important for deciding how the nodes appear when they overlap, specifically which nodes appear in front of other nodes. We can best demonstrate this functionality using figure 5.2, below.

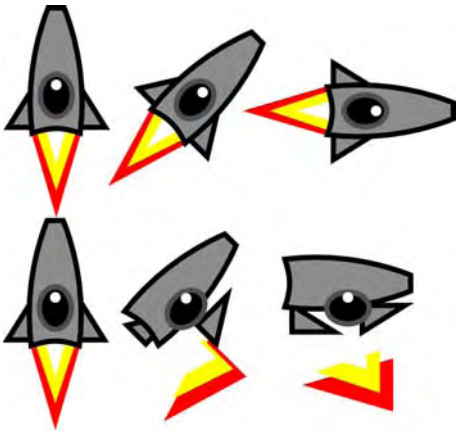


Figure 5.2 Elements in a scene graph can be manipulated without concern for how the screen will be repainted. Grouping allows manipulations to be applied consistently to multiple elements.

A space ship might be constructed from several shapes: a distorted rectangle for its body, two triangular fins, and a black circular cockpit window. It may also have a little rocket jet pointing out of its tail, likewise constructed from shapes. Each shape would be one primitive on the scene graph, one node in a tree-like structure of elements that can be rendered to screen.

5.1.2 Groups: graph manipulation made easy

Once the shapes have been added to the scene graph we can manipulate them with such transformations as a rotation. But the last thing we want is for the constituent parts to stay in the same place when rotated. The effect might be a tad unsettling if the fins appeared to fly off on an adventure all of their own (figure 5.2). We want the whole space ship to rotate consistently, as one, around a single universal origin.

Groups are the answer! They allow us to combine several scene graph elements together, so they can be manipulated as one. This includes, for example, switching on or off entire parts of the graph by flipping their group's visibility, which would be handy if we want to hide our space ship's rocket jet, as demonstrated below in figure 5.3.

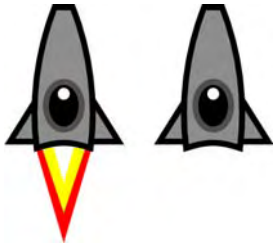


Figure 5.3 Grouping parts of the scene graph allow them to be manipulated from one convenient node in the graph, including translation, rotation, opacity (transparency) and even visibility.

This quick introductory text has only brushed the surface of the power scene graphs offer us. The retained mode approach allows sophisticated scaling, opacity (transparency), lighting and other video effects to be applied to entire swathes of objects all at once, without having to worry about the mechanics of rendering the changes to screen.

So that's all there really is to the scene graph. Hopefully your interest has been piqued by the prospect of all this pixel pushing goodness—all we need now is a suitable project to have some fun with.

5.2 Lightshow, version 1

The Eighties were a time of massive change in the computer industry. As the decade began exciting new machines, such as Space Invaders and Pac-man, were already draining loose change from the pockets of unsuspecting teenage boys, and before long video games entered the home thanks to early consoles and micro-computers. An explosion in new types of software occurred, some serious, others just bizarre.

Pioneered by the legendary llama-obsessed games programmer, Jeff Minter, Psychedelia (later, Colourspace and Trip-a-Tron) provided strange real time explosions of color on computer monitors, ideal for accompanying music. The concept would later find its way into software like Winamp and Windows Media Player, under the banner of *visualizations*.

In this chapter we're going to develop our own, very simple, *light synthesizer*. It won't respond to sound, like a real light synth should, but we'll have a lot of fun throwing patterns on screen and getting them to animate—a colorful introduction (in every sense) to the mysterious world of JavaFX's scene graph.

At the end of the project you should have a loose framework into which you can plug your own scene graph experiments. So let's jump straight in at the deep end by seeing how we plug nodes together.

5.2.1 Raindrop animations

The JavaFX scene graph API is split over many packages, specializing in various aspects of video graphics and effects. You'll be glad to know we'll only be looking at a handful of them in this chapter. At its heart, the scene graph centers around a single element know as a *node*. There are numerous nodes provided in the standard API: some draw shapes, some

act as groups, while others are concerned with layout. All the nodes are linked, at the top level, into a *stage* which provides a bridge to the outside world, be that a desktop window or a browser applet.

For our light synthesizer we're going to start by creating a raindrop effect, like tiny droplets of water falling onto the still surface of a pond (ahhh...) For those wondering (or perhaps *pondering*) how this might look, the effect is caught in action in figure 5.4, below.

Figure 5.4 Raindrops are constructed from several ripples. Each ripple expands outwards, fading as it goes.

Before we begin, it's essential we pin down exactly how a raindrop works from a computer graphics point of view:

- Each raindrop is constructed from multiple ripple circles.
- Each ripple circle animates, starting at zero width and growing to a given radius, over a set duration. As each ripple grows, it also fades.
- Ripples are staggered to begin their individual animation at regular beats throughout the lifetime of the overall raindrop animation.

Keen eyed readers will have spotted two different types of timing going on here: at the outermost level we have the raindrop activating ripples at regular beats, and at the lowest

level we have the smooth animation of an individual ripple running its course, expanding and fading. These are two very different types of animation, one digital in nature (jumping between states, with no mid-way transitions) and the other analogue in nature (a smooth transition between states), combining to form the overall raindrop effect.

5.2.2 The RainDrop class: creating graphics from geometric shapes

Now that we know what we're trying to achieve, let's look at a piece of code which defines the scene graph. Take a look at listing 5.1, below.

Listing 5.1 RainDrop.fx

```
package jfxia.chapter5;

import javafx.animation.Interpolator;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.lang.Duration;
import javafx.scene.Group;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;

package class RainDrop extends Group                                    A
{   public-init var radius:Number = 150.0;                              B
    public-init var numRipples:Integer = 3;                             B
    public-init var rippleGap:Duration = 250ms;                         B
    package var color:Color = Color.LIGHTBLUE;                          B

    var ripples:Ripple[];
    var masterTimeline:Timeline;

    init
    {   ripples = for(i in [0..<numRipples]) Ripple                     C
        {   stroke: bind color;                                         C
            animRadius: radius;                                         C
        };                                                              C
        content = ripples;
        masterTimeline = Timeline                                       D
        {   keyFrames:                                                  D
                for(i in [0..<numRipples]) KeyFrame                     D
                {   time: i*rippleGap;                                  D
                    action: function()                                  D
                    {   ripples[i].rippleTimeline                       D
                            .playFromStart();                           D
                    }                                                   D
                };                                                      D
        };                                                              D
    }                                                                   D

    package function start(x:Integer,y:Integer) : Void                  E
    {   this.translateX = x;                                            E
        this.translateY = y;                                            E
        masterTimeline.playFromStart();                                 E
    }                                                                   E
}
```

```
class Ripple extends Circle                                          F
{   var animRadius:Number;
    override var fill = null;

    def rippleTimeline = Timeline
    {   keyFrames:
        [   at (0ms)                                                 G
            {   radius => 0;                                         G
                opacity => 1.0;                                      G
                strokeWidth => 10.0;                                 G
                visible => true;                                     G
            } ,                                                      G
            at (1.5s)                                                H
            {   radius => animRadius                                 H
                    tween Interpolator.EASEOUT;                      H
                opacity => 0.0                                       H
                    tween Interpolator.EASEOUT;                      H
                strokeWidth => 5.0                                   H
                    tween Interpolator.LINEAR;                       H
                visible => false;                                    H
            }                                                        H
        ]
    };
}
```

A Subclasses Group
B External interface variables
C Multiple Ripple instances
D Timeline to activate ripples
E Starts animating ripples
F Subclasses Circle
G Start animation state
H Finish animation state

Listing 5.1 creates two classes. Together they form our desired raindrop effect on screen, with multiple circles fanning out from a central point, fading as they go. The code above will not run on its own—we need another bootstrap class, which we'll look at in a moment. For now let's consider how the raindrop effect works and how the above code implements it.

The second class, `Ripple`, implements a single animating ripple, which is why it subclasses the `javafx.scene.shape.Circle` class. Each circle is a node in the scene graph, a geometric shape which can be rendered on screen. However a raindrop with just one ripple would look rather lame. That's why the first class, `RainDrop`, is a container for several `Ripple` objects, subclassing `javafx.scene.Group` which is the standard JavaFX scene graph group node.

The `Group` works not unlike the `HBox` we encountered last chapter, except it does not impose any layout on its children. The `content` attribute is a sequence of `Node` objects which it will draw, from first to last, such that earlier nodes are drawn below later ones.

Child nodes are positioned within their parent `Group` using the `translateX` and `translateY` variables inherited from `Node`, which is the aptly named parent class of all scene graph node objects. Coordinates are local to their parent, as figure 5.5 explains. The

actual *on screen* coordinates of a given node are the sum of its own translation, plus all translations of its parent groups, both direct and indirect.

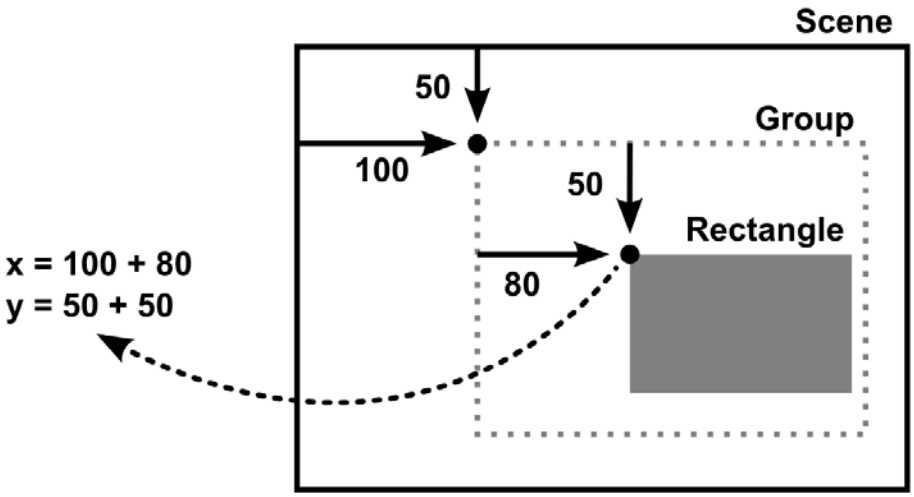


Figure 5.5 Groups provide a local coordinate space for their children.

Anyway, enough of groups, what about our code? We'll study the animation inside `Ripple` shortly, but first off we need to understand the container class, `RainDrop`, where the raindrop's external interface lies.

Firstly we define some `public-init` variables, allowing other classes to manipulate our raindrop declaratively. The `radius` is the width each ripple will grow to, while `numRipples` defines the number of ripples in the overall raindrop animation, and `rippleGap` is the timing between each ripple being instigated. Finally `color` is, unsurprisingly, the color of the ripple circles. Later on in the project we're going to manipulate the raindrop hue, so we've made `color` externally writable.

The private variable `ripples` holds our `Ripple` objects. You can see it being set up in the `init` block, then plugged into the scene graph via `content` in (parent class) `Group`.

Another private variable being set up in `init` is `masterTimeline`, which fires off each individual ripple circle animation at regular beats (controlled by `rippleGap`). The remainder of the class is a function which activates this animation. The function translates `RainDrop` to a given point, around which the ripples will be drawn, and kicks off the animation.

So I guess now all we need to know is how the animation works?

5.2.3 Timelines and animation (Timeline, KeyFrame)

The `masterTimeline` variable of the `RainDrop` class, conveniently reproduced below, deals with the outermost part of the animation.

```
masterTimeline = Timeline
{   keyFrames:
        for(i in [0..<numRipples]) KeyFrame
        {   time: i*rippleGap;
            action: function()
            {   ripples[i].rippleTimeline
                    .playFromStart();
            }
        }
};
```

Animation in JavaFX is achieved through *timelines*, as represented by the appropriately named `Timeline` class. A timeline is a duration into which points of change can be defined. In JavaFX those points are known as *key frames* (note the `KeyFrame` class reference), and they can take a couple of different forms.

The first form uses a function type to assign a piece of code to run at a given point on the timeline, while the second changes the state of one or more variables across the duration between key frames (as represented in figure 5.6).

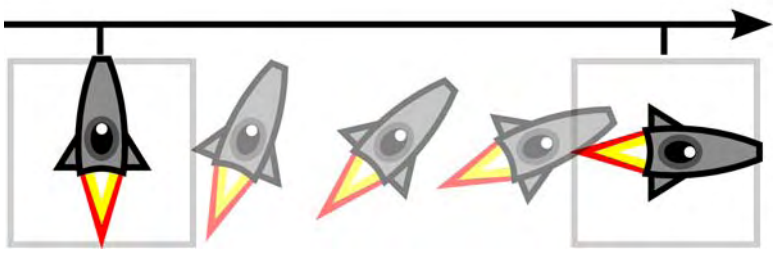


Figure 5.6 One use of key frames is to define milestones in the state of objects on the scene graph at various points throughout the duration of an animation.

In the snippet above we see only the first form in play. The `masterTimeline` is a `Timeline` object containing several `KeyFrame` objects, one for each ripple in the animation. Each key frame consists of two parts, the action to be performed (the `action`), and the point on the timeline when it should start (the `time`). The result is a timeline that, when executed, works through the `ripples` sequence calling the `start()` function on each `Ripple`, a `rippleGap` millisecond delay between each call.

As you may have guessed, this timeline controls the triggering of each ripple. Figure 5.7 shows how this works in diagrammatic form.

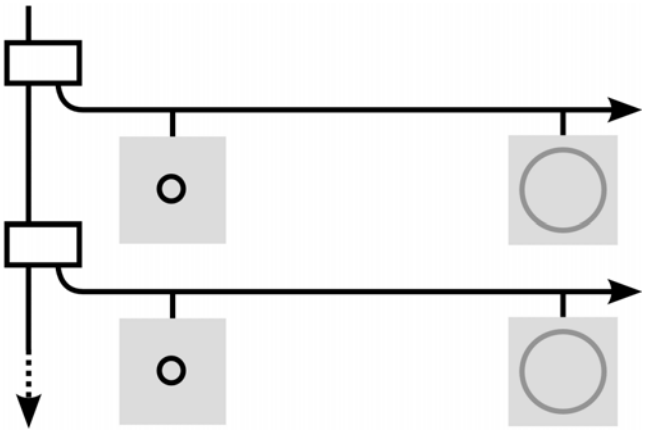


Figure 5.7 The master timeline awakes at regular intervals and fires off a single ripple timeline. The effect is a raindrop of several ripples,

The function triggers the individual ripple's growing circle animation, which uses the second form of `Timeline` to manipulate the circle over time. In the next section we'll take a look at this second, transitional, timeline form.

5.2.4 Interpolating attributes across a timeline (at, wteen, =>)

We've seen how `Timeline` and `KeyFrame` objects can be combined to call a piece of code at given points through the duration of an animation. This is a very digital like way of constructing a timeline, with actions triggered at set points along the course of the animation. But what happens if we wish to smoothly progress from one state to another?

The ripples in our animation demonstrate two forms of smooth animation effect: they grow outwards towards their maximum radius, and they become progressively fainter as the animation runs. To do this we need a different type of key frame, one which marks way-points along a journey of transition.

```
def rippleTimeline = Timeline
{   keyFrames:
    [   at (0ms)
        {   radius => 0;
            opacity => 1.0;
            strokeWidth => 10.0;
            visible => true;
        } ,
        at (1.5s)
        {   radius => animRadius
                tween Interpolator.EASEOUT;
            opacity => 0.0
                tween Interpolator.EASEOUT;
            strokeWidth => 5.0
                tween Interpolator.LINEAR;
            visible => false;
```

```
        }
    ]
};
```

Above I've reproduced the `Timeline` constructed for the `Ripple` class—it uses a very unusual syntax compared to the one we saw previously in the `RainDrop` class. You may recall earlier in this book I noted that one small part of the JavaFX Script syntax was being held over to be explained later. Now it's time to learn all about that missing bit of syntax.

The `at/tween` syntax is a shortcut to make writing `Timeline` objects easier. In effect, it's a literal syntax for `KeyFrame` objects. Each `at` block contains the state of variables at a given point in the timeline, using a `=>` symbol to match value with attribute. The duration literal following the `at` keyword is the point on the timeline to which those values will apply. Remember: those are assignments which will be made at some point in the future, when the timeline is executed—they do not take immediate effect.

Taking the above example, we can see that at 0 milliseconds the `Ripple`'s `visible` attribute is set to true, while at 1.5 seconds (1500 milliseconds) it's set to false. As invisible nodes are ignored when redrawing the scene graph, this shows the ripple at the start of the animation, then hides it at the end. We also see changes to the ripple's `radius` (from 0 to `animRadius`, making the ripple grow to its desired size), its opacity (from fully opaque to totally transparent) and its line thickness (from 10 pixels to 5.) But what about that weird `tween` syntax at the end of those lines?

The `tween` syntax tells JavaFX to perform a progressive *analogue* change, rather than a sudden *digital* change. If not for `tween`, the ripple circle would jump immediately from zero to maximum radius, fully opaque to totally transparent, and thick line to thin line, once the 1.5 second mark was reached. Tweening makes the animation run through all the stages in between.

But you'll note we do more than just move in a linear fashion from one key frame to another, we actually define how the progression happens. The constants which follow the `tween` keyword (like `Interpolator.EASEOUT` and `Interpolator.LINEAR` in the example code) define the pace of transition across that part of the animation. In our example the ease-out interpolator starts slowly and builds up speed (a kind of soft acceleration), while a linear one maintains a constant speed across the transition (no wind-up or wind-down.)

Limitations with the literal syntax

It would seem that while the '`at`' syntax is happy to accept literal durations for times, the JavaFX Script 1.0 compiler has problems with variables. This made it difficult to vary the time of a key frame using a variable. In version two of this project you'll see a slightly more verbose syntax for key frames which has the same effect as at/tween, but without this issue.

5.2.5 How the RainDrop class works

Before we move on to consider the bootstrap which will display our lovely new `RainDrop` class, I want to take a few moments to re-cap, step by step, exactly how the `RainDrop` works. We've covered quite a bit of new material in the last few pages, and it's important you understand how it all fits together.

The `RainDrop` class is a `Node` which can be rendered in a JavaFX scene graph. It holds several instances of the `Ripple` class, each of which draws and animates one circle (a ripple) in the drop animation. When the `RainDrop.start(x:Integer,y:Integer)` function is called it fires up a `Timeline`, which periodically starts the timeline inside each `Ripple`, transitioning the radius, opacity, and stroke width of the circle to make it animate.

Right, now that we have something to animate, we need to plug it into a framework to show it on screen. In the next second we'll see how that looks.

5.2.6 The LightShow class, version 1: a stage for our scene graph

To get our raindrops on screen we need to create a scene graph window, and hook the `RainDrop` class into its stage. A single raindrop wouldn't look very good, so how about we create multiple drops, which fire repeatedly as we move the mouse around? Listing 5.2 does just that!

Listing 5.2 LightShow.fx (version 1)

```
package jfxia.chapter5;

import javafx.scene.Scene;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

import java.lang.System;

Stage
{   scene: Scene
    {   def numDrops = 10;                                          A
        var currentDrop = 0;                                        A
        var lastDropTime:Long=0;                                    A
        var drops:RainDrop[];                                       A

        content:
        [   Rectangle                                               B
            {   width:400;  height:400;
                fill: Color.LIGHTCYAN;
                onMouseMoved: function(ev:MouseEvent)               C
                {   def t:Long =                                    C
                        System.currentTimeMillis();                 C
                    if((t-lastDropTime) > 200)                      C
                    {   drops[currentDrop]                          C
                            .start(ev.x,ev.y);                      C
                        currentDrop =                               C
                            (currentDrop+1) mod numDrops;           C
                        lastDropTime=t;                             C
```

```
                    }                                                    C
                }
            } ,
            drops = for(i in [0..<numDrops])
            {   RainDrop {};
            }
        ]
        width: 400;                                                      D
        height: 400;                                                     D
    }
    title: "LightShow v1";                                               E
    resizable: false;                                                    E
}
```

A Local variables, including RainDrop sequence
B The background rectangle
C Mouse event handler
D Stage dimensions
E Window configuration

The `javafx.stage.Stage` class we first encountered last chapter. Its `scene` variable is the socket into which our scene graph should be plugged, which we do using the `javafx.scene.Scene` class.

The animation works by creating several instances of the `RainDrop` class, cycling through them over and over as the mouse moves. What happens if we run out of raindops? Well, if we time things right a `RainDrop` will have finished its animation by the time it is called into service again. By knowing how long each `RainDrop` animation takes, and how frequently it is triggered, we can work out how many `RainDrop` nodes we need to continually run the animation.

Inside our `Scene` we first define some variables, local to that node:

- The `numDrops` variable defines the size of the `RainDrop` sequence—how many will be included in the scene graph. While `currentDrop` remembers which drop in the sequence is next to be animated.
- The `lastDropTime` variable records the time of the last drop animation, to ensure a reasonable gap between raindrops.
- Finally, `drops` is the `RainDrop` sequence itself.

We will manipulate these variables in an event handler. The first node in the `Scene` is a `Rectangle`, which is another kind of geometric shape just like the `Circle` class. It will give our raindrop animation a colorful backdrop. We set the dimensions of the `Rectangle` within its parent, and define a fill color. Then we assign an event handler to it.

```
onMouseMoved: function(ev:MouseEvent)
{   def t:Long = System.currentTimeMillis();
    if((t-lastDropTime) > 200)
    {   drops[currentDrop].start(ev.x,ev.y);
        currentDrop = (currentDrop+1) mod numDrops;
        lastDropTime=t;
    }
}
```

The `onMouseMoved` variable is a function type allowing us to attach some event handling code to the `Rectangle`, responding to mouse movements across its surface. The code is reproduced above (with fewer line breaks, for that bit extra readability). It works the same way as the button event handlers we saw in the Swing project last chapter, except it responds to mouse movement rather than button clicks. This event handling code is the hub of the `LightShow` class, it's here that we initiate the raindrop animation. The code is assigned to the `Rectangle` because the it covers the whole of the window interior, and as such will receive movement events wherever the mouse travels.

But how does the event handler code work?

The first thing we do is to get the current time from the computer's internal clock, using the Java method inside `java.lang.System`. We don't want to fire off new raindrops too quickly, so the next line is a check to see when we last started a fresh raindrop animation: if it's within the last 200 milliseconds we exit without further action.

Assuming we're outside the time limit, we proceed by creating a fresh raindrop animation. This requires three step: first we call `start(x:Integer,y:Integer)` on the next available raindrop, passing in the mouse event's x and y position, causing the class to begin animating around those co-ordinates. Then we move the `currentDrop` variable on to the next `RainDrop` in the sequence, wrapping around to the start if necessary—cycling through RainDrop objects as the mouse events are acted upon. Finally we store the current time, ready for the next handler invocation.

The `Rectangle` needs to access the `RainDrop` sequence from its `onMouseMoved` event handler, which is why we created a reference to the sequence as a local variable called `drops`, before we declaratively created the `Rectangle`. Having added the `Rectangle` to the `Group` we can then add `drops`, so they'll be drawn above the `Rectangle`. Because JavaFX Script is an expression language, the assignment to the `drops` variable also acts as an assignment into the enclosing scene graph sequence.

5.2.7 Running version 1

Running the code is as simple as compiling both classes, then starting up `jfxia.chapter5.LightShow` and waving your mouse over the window which appears. The effect is of circular patterns tracing the flow of your mouse, expanding and fading as they go. While they may serve no useful purpose, the application's visuals are (I hope) interesting and fun to play with. They demonstrate how rich animation can be created quite quickly from within JavaFX, with reasonably little code.

So far we've thrown a few shapes on screen and looked at how the scene graph groups things together. But the `RainDrop` is perhaps not the most efficient example of writing custom scene graph nodes. For a start, what happens if our custom node isn't a convenient subclass of an existing node? In the next section we'll see how using JavaFX's purpose made `CustomNode` class helps us create unique custom nodes. We'll also be spinning a few psychedelic shapes on screen, so if you have a lava lamp around, now would be a good time to switch it on.

5.3 Lightshow, version 2

Subclassing `CustomNode` is the recommended way of creating bespoke nodes in JavaFX. Although we managed perfectly well by subclassing `Group` for the `RainDrop`, a `CustomNode` subclass allows us a bit more control. For a start it includes a `create()` function which gets called when the node is created, acting as a lightweight constructor.

In our second version of the `LightShow` project we're going to write a `CustomNode` which will slot into the main bootstrap class, just like the `RainDrop`. We're also going to explore some exotic uses of animation timelines, to bring a splash of technicolor to our software.

5.3.1 The swirling lines animation

We'll start with a new class to create animated swirling lines, radiating out from a center point. The lines are like spokes on a wheel, spaced evenly around all 360 degrees of a ring, as in figure 5.8.

Figure 5.8 The SwirlingLines class creates a single ring of spokes, rotating around a central origin. Instances demonstrating different attribute settings are displayed.

The `SwirlingLines` class uses transformations on `Rectangle` shapes to rotate and position each line within the scene graph. All the shapes in a given ring are contained within (and controlled via) a parent group, which is in turn linked to a custom node.

As with the `RainDrop`, we'll use the class multiple times in the `LightShow`, the end effect being several nested rings of colored lines rotating in different directions, while

transitioning through several hues. Take a look at figure 5.8, above, to see what I mean. The animation will run continuously, and will not interact with the user.

So, we have a lot of interesting new ideas to cover—all we need now is some source code.

5.3.2 The SwirlingLines class: rectangles, rotations and transformations

The `SwirlingLines` source code is presented below. It contains quite a host of instance variables for configuring its operation, from line length and thickness, to the speed and direction of rotation. This will give us plenty of stuff to play with when we incorporate it into our project application, a little later on.

Above I mentioned that the lines in the finished application will continually change color. This class does not concern itself with the color changes, but it *does* bind a handy-dandy color variable, which some other class (the `LightShow` being a prime suspect) might want to manipulate. Listing 5.3 is the code.

Listing 5.3 SwirlingLines.fx

```
package jfxia.chapter5;

import javafx.animation.Interpolator;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.lang.Duration;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.transform.Transform;


package class SwirlingLines extends CustomNode
{   public-init var antiClockwise:Boolean = false;                    A
    public-init var baseAngle:Number = 0.0;                           A
    public-init var numLines:Integer = 12;                            A
    public-init var rotateDuration:Duration = 1s;                     A
    public-init var lineLength:Number = 100.0;                        A
    public-init var lineThickness:Number = 20.0;                      A
    public-init var centerRadius:Number = 20.0; zone                  A
    package var color:Color;                                          A

    var rotateSlice:Number;                                           B
    var tLine:Timeline;                                               B
    var animRotateInc:Number;                                         B


    init
    {   rotateSlice =
            (if(antiClockwise) -360.0 else 360.0) / numLines;
    }

    override function create():Node
```

```
    {   def node = Group
        {   content: for(i in [0..<numLines])
            {   Rectangle
                {   width: lineLength;                                  C
                    height: lineThickness;                              C
                    fill: bind color;                                   C
                    transforms:                                         D
                    [   Transform.rotate                                D
                        (   baseAngle + rotateSlice*i ,                 D
                            0,0                                         D
                        ) ,                                             D
                        Transform.translate                             D
                        (   centerRadius ,                              D
                            0-lineThickness/2                           D
                        )                                               D
                    ];                                                  D
                };
            };
            rotate: bind baseAngle + animRotateInc;                     E
        };

        // Animation timeline
        tLine = Timeline
        {   repeatCount: Timeline.INDEFINITE;                           F
            keyFrames:
            [   KeyFrame                                                G
                {   time: 0s;                                           G
                    values:                                             G
                    [   animRotateInc => 0.0                            G
                    ];                                                  G
                } ,                                                     G
                KeyFrame                                                H
                {   time: rotateDuration;                               H
                    values:                                             H
                    [   animRotateInc => rotateSlice                    H
                           tween Interpolator.LINEAR                    H
                    ];                                                  H
                }                                                       H
            ];
        };
        tLine.play();                                                   I

        return node;
    }
}
```

A External attributes
B Internal attributes, mainly animation
C Bound to external attrs.
D Transformation ops array
E Rotate Group
F Run forever
G Start of rotation
H End of rotation
I Start animation

The `SwirlingLines` class shown in listing 5.3 is a custom node, as denoted by its subclassing of the `javafx.scene.CustomNode` class. While subclassing `Group` was fine

for our simple ripples, we need something a bit more powerful for the effect we're building. `CustomNode` subclasses have the luxury of a `create()` function acting as a kind of lightweight constructor, returning the `Node` object to be added to the scene graph. Figure 5.9 is a diagrammatic rendering of what the code produces.

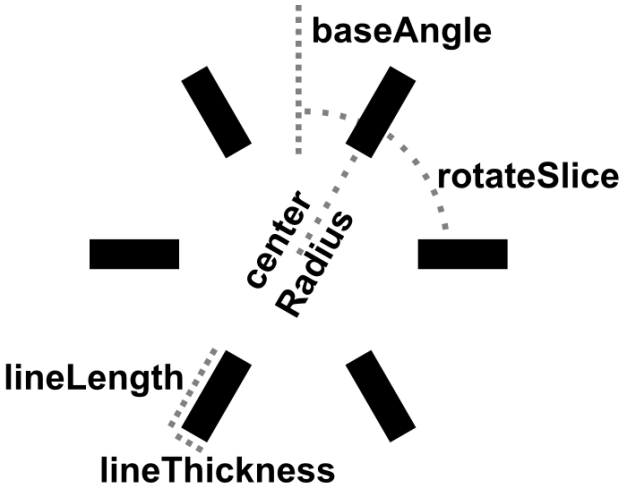


Figure 5.9 SwirlngLines creates a ring of rectangles, fully customizable from a host variables.

Okay, taking a look at the class we can clearly see several instance variables available for tailoring the class declaratively. Figure 5.9 shows the main initialization variables in diagram form, but here's a description of what they do:

- The `color` variable controls the lines color (obviously!), while `lineLength` and `lineThickness` control the size of the lines. As we'll be manipulating `color` throughout the run of the application it has been made `package` visible, and its reference in the scene graph is bound.
- The `antiClockwise` flag determines the direction of animation, while `baseAngle` controls how the lines are initially orientated (the first line doesn't have to start at zero degrees).
- The `numLines` determines how many lines form the ring—they will be evenly spaced around the total 360 degrees.
- The `rotateDuration` attribute controls how long it takes for the ring to perform one animation cycle. If there are 16 lines, this will be the time it takes to animate through one 16th of a total 360 degree revolution.
- Finally, `centerRadius` details the empty space between the rotation center and

the inner end of each line. In other words, how far away from the center the lines are moved.

There are also some private, internal mechanics, variable.

- The `rotateSlice` value is the angle between each line in the ring. This is 360 divided by the number of lines. The value is used to constrain the animation, which we'll look at in a moment.
- The `tLine` is our `Timeline` object, and `animRotateInc` is the value we change during the timeline animation.

The rotation is performed at the group level, thanks to the group node's `rotate` variable being bound to `animRotateInc`. Although the ring will appear to rotate freely through a full 360 degrees, this is actually an optical illusion. The animation only moves between lines, so if there are four lines, our animation will continually run between just 0 and 90 degrees.

The next part of the class is a function called `create()`, which returns a `Node`. The `CustomNode` class provides this function specifically for us to override with code to build our own scene graph structure (note the `override` keyword). The node we return from this function will be added to the scene graph, which is what we'll look at next.

5.3.3 Manipulating node rendering with transformations

The `create()` function does all the work of setting up the nodes in the scene graph, and defining the animation timeline. The first of these two responsibilities involves creating a sequence of `Rectangle` objects, the inner code for which is reproduced below with fewer line breaks:

```
Rectangle
{   width: bind lineLength;
    height: bind lineThickness;
    fill: bind color;
    transform:
    [   Transform.rotate(baseAngle + rotateSlice*i, 0,0) ,
        Transform.translate(centerRadius , 0-lineThickness/2)
    ];
};
```

The function uses a `for` loop to populate a `Group` with these `Rectangle` objects. Width, height and fill color are all tied to the external attributes of the class. So far, nothing new—but take a look at the `transform` assignment, what's going on there?

Transformations are discrete operations applied to a node (and thereby its children) during the process of rendering it to screen. The `javafx.scene.transform.Transform` class contains a host of different transformation functions which can be applied to your nodes. Transformations are executed in order, from first to last, and the order in which they are applied is often crucial to the end result. Figure 5.10 can be used to explain this.

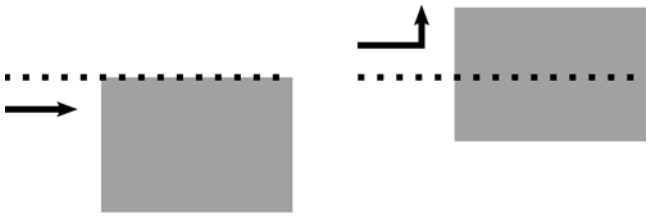


Figure 5.10 With and without centering: moving the Rectangle negatively in the y axis, by half its height, has the effect of centering it around its origin—in this case the radial spoke of a ring.

Let's consider the two operations in our example code: first we perform a rotation of a given angle around a given point, then we move (translate) the node a given distance in the x and y directions. The first operation ensures the line is drawn at the correct angle, rotated around the origin (which is the center of the ring, recall). The second operation moves the line away from the origin, but also centers it along its radial by moving *up* half its height. I realize this might be a little hard to visualize in your head, so take a look at figure 5.10, which hopefully will clarify what's happening.

When up sometimes means down

When I say “up”, it's a relative term; up in relation to the rotation we previously applied. If thinking about this hurts your head, try to picture it this way: the screen is a giant wall, and tacked onto that wall are separate pieces of paper with drawings on them (like you'd find in a junior school classroom, where the kids have their crayon masterpieces on show).

We can take any one of these pieces of paper and rotate in on the wall—for example we could display a given picture upside-down. Consider this: if we take one of the drawings down off the wall and erase an object, redrawing it higher up (like we move the Sun further up in the sky), then we tack the drawing back onto the wall upside-down, did we move the object up or down? In terms of the paper, we moved it up. But after we apply the rotation we moved it down in terms of the overall wall. We have two co-ordinate spaces in operation here, the global one (the wall) and the local one (the paper).

To return to our project source code, when we said we moved the `Rectangle` “up” we meant in terms of its local co-ordinate space. That space (like the paper on the wall) is rotated, but it doesn't matter, because from a local point of view up still means up (left still means left, etc.) even if the rotation actually changes the effect in terms of the global space.

As previously noted, it's important to consider the order in which transforming operations are performed. Turning 45 degrees then walking forward five paces is not the same as

walking forward five paces then turning 45 degrees–check out figure 5.11 if you don't believe me. Always remember the golden rule: each time a node is drawn its transformations are applied in order, from first to last.

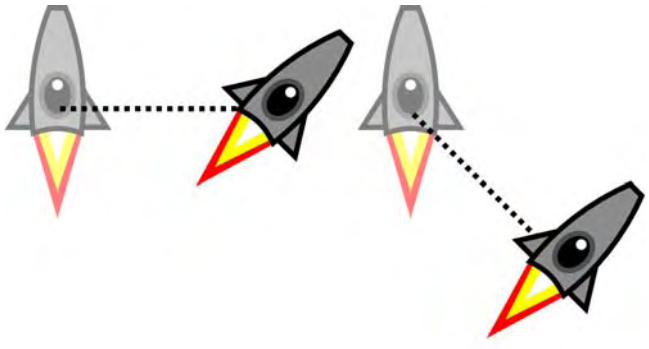


Figure 5.11 Two examples of transformations: translate then rotate (left), and rotate then translate (right). The order of the operations results in markedly different results.

Now that we understand transformations, let's look at the remainder of the code:

```
tLine = Timeline
{   repeatCount: Timeline.INDEFINITE;
    keyFrames:
    [   KeyFrame
        {   time: 0s;
            values:
            [   animRotateInc => 0.0
            ];
        } ,
        KeyFrame
        {   time: rotateDuration;
            values:
            [   animRotateInc => rotateSlice
                   tween Interpolator.LINEAR
            ];
        }
    ];
};
tLine.play();
```

The above creates a timeline which runs continually once started, thanks to `Timeline.INDEFINATE`, with two `KeyFrame` objects, one marking its start and the other its end. All the timeline does is tween the attribute `animRotateInc`, which the `Group` binds to. Changes to this attribute cause the `Group`, and its `Rectangle` contents, to rotate.

The timeline above look a little different to the `at` syntax we saw in action in the `Ripple` class. There's actually no difference in terms of functionality, we're just writing out the `KeyFrame` objects long hand instead of using the briefer syntax. The `KeyFrame` code is

quite easy to define declaratively: `time` is obviously the point at which the key frame should be active, while `values` is a comma separated list (a sequence) of *attribute => value* definitions.

After we've created the timeline we kick it off immediately by calling its `play()` function. The Timeline class has a number of functions for controlling playback. The two we've seen in this project are `playFromStart()` and `play()`. The former restarts a timeline from the beginning, while the latter picks up where it left off. As our above timeline will run indefinitely we can use either.

> **Limitations with the literal syntax, part two**
>
> Mentioned previously, the 'at' syntax for describing key frames seems not to like variables, only time literals. At least that seems to be the case with the v1.0 compiler. However, the verbose syntax above doesn't suffer from the same problems.

5.3.4 The LightShow class, version 2: color animations

Now we have a swirling lines class, let's add it into our application class, `LightShow`. We also want to create some kind of psychedelic color effect as well. Listing 5.4 show how the updates change the code:

Listing 5.4 LightShow.fx (version 2)

```
package jfxia.chapter5;

import javafx.animation.Interpolator;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.lang.Duration;
import javafx.scene.Scene;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

import java.lang.System;


def sourceCols =                                                    A
[    "#ff3333","#ffff33","#33ff33",                                 A
     "#33ffff","#3333ff","#ff33ff"                                  A
];                                                                  A
def colorShifts = for(i in [0..<6])                                 A
{   ColorShifter                                                    A
    {   sourceColors: sourceCols;                                   A
        duration: 3s;                                               A
        offset: i;                                                  A
    };                                                              A
}                                                                   A
```

```
def sceneWidth:Number = 400;
def sceneHeight:Number = 400;
Stage
{   scene: Scene
    {   def numDrops = 10;
        var currentDrop = 0;
        var lastDropTime:Long=0;
        def drop = for(i in [0..<numDrops])
        {   RainDrop {};
        }

        def lines = for(i in [0..<6])                                   B
        {   SwirlingLines                                               B
            {   def ii = i+1;                                           B
                translateX: sceneWidth/2;                               B
                translateY: sceneHeight/2;                              B
                numLines: 6+i;                                          B
                color: bind colorShifts[i].color;                       B
                centerRadius: (ii)*20;                                  B
                lineLength: (ii)*10;                                    B
                lineThickness: (ii)*3;                                  B
                antiClockwise: ((i mod 2)==0);                          B
                rotateDuration: 1s/(ii);                                B
            }                                                           B
        };                                                              B

        def rect = Rectangle
        {   width: sceneWidth;
            height: sceneHeight;
            fill: Color.BLACK;
            onMouseMoved: function(ev:MouseEvent)
            {   def t:Long =
                    System.currentTimeMillis();
                if((t-lastDropTime) > 200)
                {   drop[currentDrop].color =                           C
                        colorShifts[0].color;                           C
                    drop[currentDrop]
                        .start(ev.x,ev.y);
                    currentDrop =
                        (currentDrop+1) mod numDrops;
                    lastDropTime=t;
                }
            }
        };

        content:
        [   rect ,
            lines ,                                                     D
            drop
        ]
        width: sceneWidth;
        height: sceneHeight;
    }
    title: "LightShow v2";
    resizable: false;
    onClose: function() { FX.exit(); }                                  E
}
```

```
class ColorShifter
{   public-init var duration:Duration = 3s;                                F
    public-init var sourceColors:String[];                                 F
    public-init var offset:Integer=0;                                      F

    var color:Color;                                                       G
    var tLine:Timeline;

    init
    {   def gap = duration / ((sizeof sourceColors)-1);
        tLine = Timeline
        {   def arrSz = sizeof sourceColors;
            repeatCount: Timeline.INDEFINITE;
            keyFrames: for(i in [0..<arrSz+1])                             H
            {   KeyFrame                                                   H
                {   def pos = (offset+i) mod arrSz;                        H
                    time: gap * i;                                         H
                    values: color =>                                       H
                        Color.web(sourceColors[pos])                       H
                        tween Interpolator.LINEAR;                         H
                }                                                          H
            }                                                              H
        };
        tLine.play();
    }
}
```

A Create seven color animations
B Swirling lines sequences, declaratively defined
C Color shift our raindrops
D Lines added to scene graph
E Explicitly close window
F Declaration variables
G Output color
H A KeyFrame for each color, wrapped around

We'll have a look at the color shifter first. At the start of listing 5.4 there's a sequence of colors called `sourceCols`, using web style definitions (`#rrggbb`, as hex.) Following that, a `for` loop creates seven `ColorShifter` objects.

The `ColorShifter` provides us with an every shifting color, cycling through a collection of shades over a period of time. You can find its code at the bottom of listing 5.4. The external interface is as follows:

- The `duration` attribute is the time it will take to do one *full-circle* of the colors.
- The sequence `sourceColors` provides the hues to cycle through, and `offset` is the index to use as the first color.
- Finally, `color` is the output—the current hue.

The class creates a timeline with `KeyFrames` for each color, using tweening to ensure a smooth transition between each. The first color is used twice, at both ends of the animation, to ensure a smooth transition when wrapping around from last to first color. That's why the loop runs for one greater than the actual size of the source sequence and the `mod` operator is

applied to the loop index to keep it within range. Once created the `ColorShifter` timeline is started, and runs continually.

So, that's the color animations. Returning to the scene graph, let's have a look at how the `SwirlingLines` are added to our `Group` node:

```
def lines = for(i in [0..<6])
{   SwirlingLines
    {   def ii = i+1;
        translateX: bind fWidth/2;
        translateY: bind fHeight/2;
        numLines: 6+i;
        color: bind colorShifts[i].color;
        centerRadius: (ii)*20;
        lineLength: (ii)*10;
        lineThickness: (ii)*3;
        antiClockwise: ((i mod 2)==0);
        rotateDuration: 1s/(ii);
    }
 };
```

Nothing particularly unusual here. We create seven rings of lines, and each is bound to a different ever-changing `ColorShifter`. Because the `ColorShifter` objects were all declared with using different source colors (we used a different `offset` each time) each ring pulses with a different part of the `sourceCols` input sequence. Each ring has more lines, longer and thicker, with a larger gap at the center. The rings alternate clockwise and anticlockwise, and with outer rings rotating faster than inner ones.

Now that we have the code, let's see what happens when we run it...

5.3.5 Running version 2

Version two adds rotating patterns of lines and ever changing colors, all animating away merrily without us having to get involved in any ugly code to draw them on screen, as we'd need to if this was a Swing application using immediate mode rendering.

Figure 5.12 shows the application running.

Figure 5.12 Version two of the project application, featuring both swirling lines and raindrops.

When you run the code you'll see I added a few extra changes to version one: the background is now black, and the raindrops are bound to a `ColorShifter` too. The effect is an explosion of color patterns across the window as the mouse is moved.

5.4 Lost in translation? Positioning nodes in the scene graph

Before we round off this chapter there's one topic we really *should* review: the relationship between a scene graph node's translation (`translateX` and `translateY`), and its coordinates. It's vital you understand how the two work with each other.

Each scene graph node has a way of specifying its location, for example `Rectangle` has `x` and `y`, `Circle` has `centerX` and `centerY`, and so on. These define points within the node's own coordinate space, quite separate to (although ultimately combined with) its translation in the graph as a whole. See figure 5.13, below.

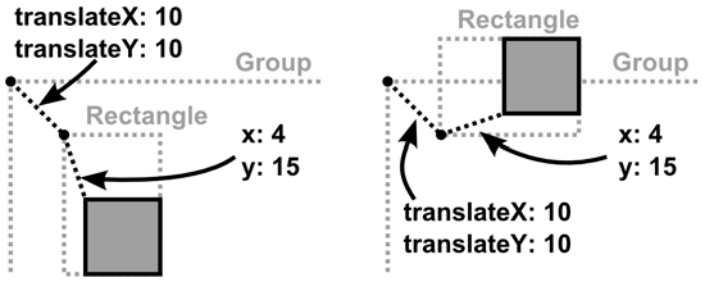


Figure 5.13 A Rectangle with its own local coordinates, translated within a Group. The local coordinates are not affected when the node is transformed in its parent's space, like a rotation by 270 degrees,.

To borrow section 5.3.3's paper/wall analogy if each shape is a drawing on a piece of paper tacked onto a wall, we might say `Circle.centerX` and `Circle.centerY` represent the circle's position within the paper, while `Circle.translateX` and `Circle.translateY` represent the paper's position within the wall. The former is the shape's location within its own space; the latter is its location within its parent's space. The important point is this: the node's local space is unchanged, no matter how the node is rotated, bent, folded, or otherwise manipulated in the scene graph outside.

5.5 Summary

In this chapter we took our first look at the scene graph, and played around with throwing shapes on screen. We saw multiple examples of timeline based animation, and explored how to define timelines to suit different purposes: triggering events at given moments, and progressively transitioning variables between different states. We also witnessed how timelines could be used to animate more than just shapes on screen. And to cap it all we dabbled with mouse events. Whew!

The `LightShow` example isn't the most useful application in the World, we didn't even wire it up to a sound source like true *visualizations*, but I hope you found it suitably entertaining. You can use the `LightShow` application as a framework for plugging in and trying your own `CustomNode` experiments, if you wish. There's plenty of different transformations we didn't have space to cover in this chapter—you might want to try playing with some of them, distorting the `Rectangle` lines or even adding different shapes of your own, and seeing what effects they create.

In the next chapter we'll be staying with the scene graph, but looking at building a slightly more useful application (using video, no less!) We'll also be creating our own custom UI components and looking at some of the effects we can do using the scene graph. So by all means have fun experimenting with the `LightShow`, and when you're ready I'll see you in the next chapter...

6

Moving pictures

In previous chapters we developed an application using JavaFX's Swing wrappers, and played around with the scene graph. Now it's time to turn our hand towards creating a UI that combines the functionality of a traditional desktop application with the visual pizazz of the new scene graph library. And when it comes to desktop software, you don't often get showier than media players.

In the version 1.1 release of JavaFX, used while writing this chapter, JavaFX's own library of UI *controls* was largely still in development. The library promises customization through such things as CSS like style documents. Until it arrives (hopefully by the time you read this) the Swing equivalents will provide a suitable stand in. But even after it arrives, there will always be cases when we need to achieve a very individual type of functionality or a particular animation effect, not supported by existing libraries. It's at times like these we find ourselves forced to roll our own widgets, taking complete control of how a widget behaves and animates. In this chapter we'll be looking at writing simple UI widgets using only the scene graph nodes, just to see how it's done.

We'll also look at using the media classes to play videos from the local hard disk. To try out the project (see figure 6.1, below) you'll need a directory of movie files, such as MPEG and WMV files.

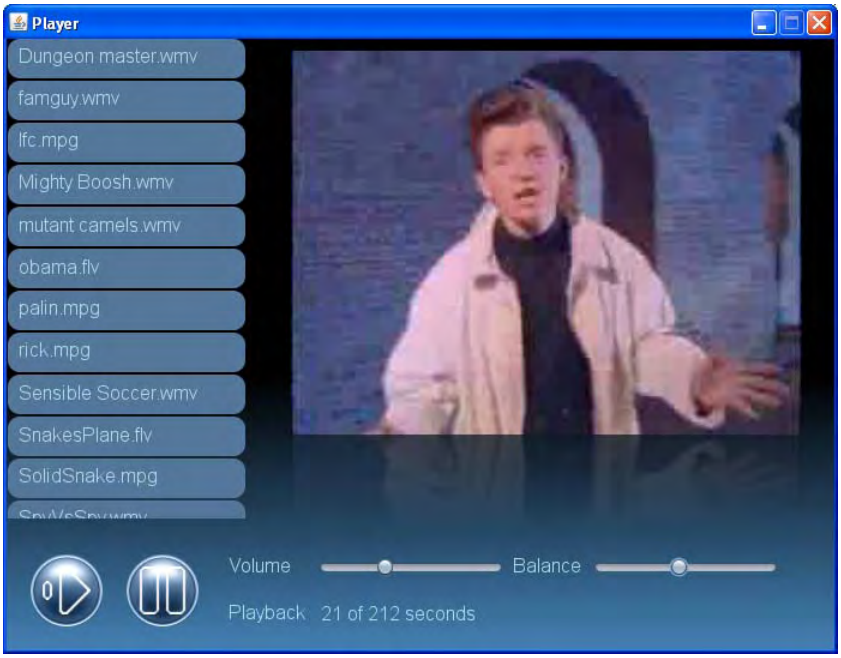


Figure 6.1 A preview of the simple video player we'll be building in this chapter.

The video player we'll be developing is very primitive, lacking much of the functionality of full sized professional players like Window Media Player or Quicktime. A full player application would have been ten times the size with little extra educational value. Although primitive in function, our player will require a couple of bespoke widgets, each demonstrating a different technique or skill.

We'll also be adding a gradient fill to the background, and a real time reflection effect will be applied to the video itself, making it look like it's resting on a shiny surface.

DOWNLOAD NEEDED

This project requires a few images, which can be downloaded along with the source code from the book's web site: `http://www.manning.com/morris/`

Over the coming pages you'll learn about working with images and video, creating widgets from scene graph shapes, and applying fancy color fills and effect. This is quite a busy project, with a lot of interesting ground to cover, so let's get started.

6.1 The Video Player, version 1

In this version of the player software we're focusing mainly on building the user interface controls we'll need when we manipulate our video later on. Before JavaFX came along getting video to work under Java would have deserved an entire book to itself. Thankfully JavaFX makes things a lot easier. JavaFX's video classes are easy to use, so we don't have to devote the entire chapter to just getting video on screen.

You can see what this stage of the project looks like in figure 6.2, below.

Figure 6.2 The interface for version one of our application.

We'll begin simply enough, with the control panel which sits at the foot of the video player. It includes two examples of custom UI classes. The first is an image button, demonstrated to the left figure 6.2; the second is a layout node, positioning the sliders and text to the right of figure 6.2.

We'll tackle the button first.

6.1.1 The Util class: creating image nodes

As before, the code is broken up into several classes and source files. But before we look at the button class itself, we'll take a short detour to consider a utility class. Quite often when we build software the same jobs seem to keep coming up time and again, and sometimes it's useful to write small utility functions which can be called upon by other parts of the code.

In our case the bespoke button we're writing needs to load images from a directory associated with the program—these are not images the user would choose, but icons which come bundled with our player application. The code is show in listing 6.1.

Listing 6.1 Util.fx

```
package jfxia.chapter6;

import javafx.scene.image.Image;

import java.io.File;
import java.net.URL;


package function appImage(f:String) : Image
{   Image
    {   url: (new URL("{__DIR__}../../images/{f}")).toString();
```

```
        };
    }
```

The script level (static) function `appImage()` is used to load an application image from the project's `images` directory, such as icons, backgrounds, and other paraphernalia which might constitute our user interface. It accepts a string—the filename of the image to load—and returns a JavaFX `Image` class representing that image. The JavaFX `Image` class uses a URL as its source parameter, explaining the presence of the Java `URL` class. Hopefully you've noticed a strange symbol in the middle of the code: `__DIR__`. What does it do?

Well, it's actually an example of a pre-defined variable for passing environment information into our running program.

- `__DIR__` returns the location of the current class file, as a URL. It may point to a directory if the class is merely a regular `.class` bytecode file, or it may point to a JAR file if the class lives inside a Java archive.
- `__FILE__` returns the full filename and path of the current class file, as a URL.

Note how both `__FILE__` and `__DIR__` are URLs instead of files. If the executing class lives on a local drive the URL will use the 'file:' protocol. If the class was loaded from across the internet it may take the form of a 'http:' based address.

Hopefully most of you will have realized the `appImage()` function isn't the most robust piece of code in the World. It relies on the fact that our classes live in a package called `jfxia.chapter6`, which resolves to two directories deep from the application's root directory. It backtracks out of those directories and looks for an `images` directory living directly off the application root. If we were to package the whole application up into a JAR this brittle assumption would break. But for now it's enough to get us going.

6.1.2 The Button class: scene graph images and user input

The Button class creates a simple push button of the type we saw in our Swing example, back in chapter four. Except this one is built entirely using the scene graph, and has a little bit of animation magic when it's pressed. The `Button` is a very simple component, which offers an ideal introduction to creating our own UI widgets.

The button is constructed from two bitmap graphics: a background frame and a foreground icon. When the mouse moves over the button its color changes, and when clicked it changes again, requiring three versions of the background image: the *idle* mode image, the *hover* image and the *pressed* (clicked) image. See figure 6.3 below

Figure 6.3 The button is constructed from two bitmap images: a background (blue circle) and icon (arrow). When the button is clicked a 'ghost' of its icon expands and fades.

When clicked, the copy of the icon expands and fades, creating a pleasing ghosted zoom animation. Figure 6.3 demonstrates the effect: the arrow icon animates, over the blue circle background. We'll be constructing the button from scratch, using a `CustomNode`, and implementing its animation as well as providing our own event handlers (because a button which stayed mute when pressed is about as much use as the proverbial chocolate teapot).

Anyway, enough theory—now let's look at some code. Listing 6.2 is below.

BROKEN LISTINGS

Because the button we're developing is a little more involved than the code we've seen thus far, I've broken its listing into two chunks, with accompanying text.

Listing 6.2 Button.fx (part 1)

```
package jfxia.chapter6;

import javafx.animation.Interpolator;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.lang.Duration;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.input.MouseEvent;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.shape.Rectangle;


var backIdleIm:Image;                                                    A
var backHoverIm:Image;                                                   A
var backPressIm:Image;                                                   A

package class Button extends CustomNode
```

```
{   public-init var iconFilename:String;                                    B
    public-init var clickAnimationScale:Number = 2.5;                       B
    public-init var clickAnimationDuration:Duration = 0.25s;                B
    public-init var action:function(ev:MouseEvent);                         B

    var foreImage:Image;                                                    C
    var backImage:Image;                                                    C
    var maxWidth:Number;                                                    C
    var maxHeight:Number;                                                   C

    var animButtonClick:Timeline;                                           D
    var animScale:Number = 1.0;                                             D
    var animAlpha:Number = 0.0;                                             D
    var iconVisible:Boolean = false;                                        D


    init
    {   if(backIdleIm==null)                                                E
        {   backIdleIm =                                                    E
                Util.appImage("button_idle.png");                           E
            backHoverIm =                                                   E
                Util.appImage("button_high.png");                           E
            backPressIm =                                                   E
                Util.appImage("button_down.png");                           E
        }                                                                   E
        foreImage = Util.appImage(iconFilename);                            E
        backImage = backIdleIm;                                             E
        maxWidth = java.lang.Math.max                                       E
        (   foreImage.width*clickAnimationScale ,                           E
            backImage.width                                                 E
        );                                                                  E
        maxHeight = java.lang.Math.max                                      E
        (   foreImage.height*clickAnimationScale ,                          E
            backImage.height                                                E
        );                                                                  E
    }                                                                       E
    // ** Part two below
```

A Button frame images
B External interface variables
C Private variables
D Animation variables
E Which is bigger image?

Take a look at `Button.fx` in listing 6.2 above. It covers the first part of our custom button, including the variable definitions. The first thing we see is some script variables to use as the button background images. As these are common to all instances of our `Button` class we save a few bytes by loading them just the once.

Our `Button` class extends javafx.scene.`CustomNode`, which is the recognized way to create new scene graph nodes from scratch. Inside the class itself we find several external interface variables, used to configure its behavior:

- The `iconFilename` variable holds the filename of the icon image.
- The `clickAnimationScale` and `clickAnimationDuration` variables control the size and timing of the fade/zoom effect.

- The `action` function type is for a button press event handler.

There's also some internal implementation variables:

- The `foreImage` and `backImage` are the button's current background and foreground images.
- Variables `maxWidth` and `maxHeight` figures out how big the button should be, based upon whichever is larger, the foreground or the background image.
- The variables `animButtonClick`, `animScale`, `animAlpha` and `iconVisible` all form part of the fade/zoom animation.

We use an `init` block to set up some of the internal variables, based upon the `public-init` variables used. As `init` runs after the class variables have been set, and the `public-init` variables cannot be changed from outside the class once initialized, we can safely fix the dimensions of the button from its images. But first, we check whether the *static* images have been loaded (if this is the first ever `Button` instantiated, they won't be) and load them if necessary.

Taking control

We subclassed from `javafx.scene.CustomNode`, which is how we should create our own nodes from scratch. However, I mentioned at the start of the chapter JavaFX is destined to have its own UI widget library. This library wasn't fully developed in the debut JavaFX release used to write this book, but hopefully it won't be delayed too long.

The base class for these JFX 'controls' is `javafx.scene.control.Control`, which simply adds standardized support for skinning to custom nodes. If support for skins and styling is important in your own UI code, take a look at the `javafx.scene.control` package.

Are you ready for the second half of the code? Listing 6.3 is where we create our button's scene graph.

Listing 6.3 Button.fx (part 2)

```
// ** Part one above
override function create() : Node
{   def n = Group
    {   translateX: maxWidth/2;
        translateY: maxHeight/2;
        content:
        [   Rectangle                                                    F
            {   x: 0-(maxWidth/2);
                y: 0-(maxHeight/2);
                width: maxWidth;
                height: maxHeight;
                opacity: 0;
            } ,
```

```
        ImageView                                                       G
        {   image: bind backImage;
            x: 0-(backImage.width/2);
            y: 0-(backImage.height/2);
            onMouseEntered: function(ev:MouseEvent)                     H
            {   backImage = backHoverIm;                                H
            }                                                           H
            onMouseExited: function(ev:MouseEvent)                      I
            {   backImage = backIdleIm;                                 I
            }                                                           I
            onMousePressed: function(ev:MouseEvent)                     J
            {   backImage = backPressIm;                                J
                animButtonClick.playFromStart();                        J
                if(action!=null)  action(ev);                           J
            }                                                           J
            onMouseReleased: function(ev:MouseEvent)                    K
            {   backImage = backHoverIm;                                K
            }                                                           K
        } ,
        ImageView                                                       L
        {   image: foreImage;
            x: bind (0-foreImage.width)/2;
            y: bind (0-foreImage.height)/2;
            opacity: bind 1-animAlpha;
        } ,
        ImageView                                                       M
        {   image: foreImage;
            x: bind 0-(foreImage.width/2);
            y: bind 0-(foreImage.height/2);
            visible: bind iconVisible;
            scaleX: bind animScale;
            scaleY: bind animScale;
            opacity: bind animAlpha;
        }
    ]
};

animButtonClick = Timeline                                              N
{   keyFrames:
    [   KeyFrame
        {   time: 0s;
            values:
            [   animScale => 1.0  ,
                animAlpha => 1.0 ,
                iconVisible => true
            ]
        } ,
        KeyFrame
        {   time: clickAnimationDuration;
            values:
            [   animScale => clickAnimationScale
                    tween Interpolator.EASEOUT ,
                animAlpha => 0.0
                    tween Interpolator.LINEAR ,
                iconVisible => false
            ]
        }
```

```
                ];
            };
            return n;
        }
    }
```

F Force dimensions
G Background image
H Mouse enters node
I Mouse leaves node
J Mouse button down
K Mouse button up
L Icon image
M Animation image
N Animation timeline

At the top of listing 6.3 is the `create()` function where the actual scene graph is constructed. This is a function inherited from `CustomNode`, which is why `override` is present, and is the recognized place to build your graph.

As we've come to expect, the various elements are held in place with a `Group` node. Translating the button's x and y co-ordinate space (`translateX` and `translateY`) into the center makes it easier to align the various elements of the button. The `Group` is constructed from one `Rectangle` and three `ImageView`'s (figure 6.4). The `Rectangle` is invisible and merely forces the dimensions of the button to its maximum required size to prevent resizing (and jiggling neighboring nodes around) during animations.

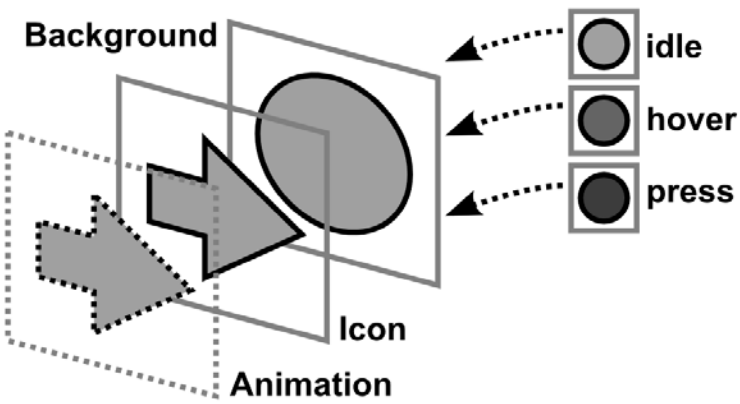


Figure 6.4 Ignoring the invisible Rectangle (used for sizing) there are three layers in our button.

In front of the rectangle there's three `ImageView` objects. What's an ImageView?

It's yet another type of scene graph node, this one displays `Image` objects—the clue is in the class name. Our button requires three images (see figure 6.4):

- The button background image, which changes when the mouse hovers over or pressed the button.
- The icon image displays the actual button symbol.

- The animation image is used in the fade/zoom effect when the button is pressed. This is a copy of the icon image, hidden when the animation isn't playing.

Take a look at the code for the first `ImageView` declaration. Like other `Node` subclasses, the `ImageView` can receive events, and has a full complement of event handling function types into which we can plug our own code. In the case of the background `ImageView` we've wired up handlers to change its image when the mouse rolls into or out of the node, and when the mouse button is pressed and released. The button press is by far the most interesting handler, as it not only changes the image, but launches the fade/zoom animation and calls any `action` handler which might be linked to our `Button` class.

The second `ImageView` in the sequence just displays the regular icon—the only cleverness is it will fade into view as the animating fade/zoom icon fades out. Subtracting the current animation opacity from 1 means this image always has the opposite opacity to the animation icon image., so as the zooming icon fades out of view the regular icon reappears, creating a pleasing *full circle* effect.

The third, and final, `ImageView` in the sequence is the animation icon. It performs the fabled fade/zoom, and as you'd expect it's heavily bound to the object variables which are manipulated by the animation's time line.

And talking of time lines, the `create()` function is rounded off by a classic start/finish key frame example, not unlike the examples we saw last chapter. The animation icon's size (scale) and opacity (alpha) are transitioned, while the animation `ImageView` is switched on at the start of the animation and off at the end. Simple stuff!

And so that, ladies and gentlemen, boys and girls, is our `Button` class. It's not perfect, (indeed it has one minor limitation we'll consider later, when plugging it into our application) but it shows what can be achieved with only a modest amount of effort.

6.1.3 The GridBox class: layout your nodes

Our button class is ready to use. Now it's time to turn our attention to the other piece of bespoke UI code in this stage of the application: the layout node.

> **WARNING: MAJOR ROADWORKS AHEAD**
>
> As you probably know, this book was written against JavaFX 1.1 (December 2009), and at the time of writing much of JavaFX's own UI controls were still being finalized. This section shows an example of a layout node using JavaFX 1.1; but later JavaFX developments will likely supersede its implementation, and some of the internal details it relies upon will change. Already in v1.0 the foundations were being laid for what's to come, with classes like `javafx.scene.layout.Resizable` allowing nodes to advertise their sizing requirements. This `GridBox` class should therefore be treated as a stop-gap, until an *official* alternative is ready.

Figure 6.5 shows the effect we're after, the text and slider nodes in that screen shot are held in a loose grid, with variable sized columns and rows which shrink to the width or height of their contents.

Figure 6.5 The GridBox node positions its children into a grid, with flexible column and row sizes.

This is not an effect we can easily construct with the `javafx.scene.layout.HBox` and VBox classes we encountered a couple of chapters back (you'll recall they arrange their contents in a line, either horizontally or vertically), so we have no option but to write our own layout node. Listing 6.4 does just that.

Listing 6.4 GridBox.fx

```
package jfxia.chapter6;

import javafx.scene.Group;
import javafx.scene.Node;

package class GridBox extends Group
{   public-init var columns:Integer = 5;                                    A
    public-init var centerX:Boolean = false;                                B
    public-init var centerY:Boolean = false;                                B
    public-init var horizontalGap:Number = 0.0;                             C
    public-init var verticalGap:Number = 0.0;                               C

    override var content on replace                                         D
    {   impl_requestLayout();                                               D
    }                                                                       D

    init                                                                    E
    {   impl_layout = doGridBoxLayout;                                      E
        impl_requestLayout();                                               E
    }                                                                       E

    function doGridBoxLayout(g:Group) : Void
    {   def sz:Integer = sizeof content;
        var rows:Integer = (sz/columns);
        rows += if((sz mod columns) > 0) 1 else 0;

        var colSz:Number[] = for(i in [0..<columns]) 0.0;
        var rowSz:Number[] = for(i in [0..<rows]) 0.0;

        for(i in [0..<sz])                                                  F
        {   def c:Integer = (i mod columns);                                F
            def r:Integer = (i / columns).intValue();                       F
            def n:Node = content[i];                                        F
            if(n.layoutBounds.width > colSz[c])                             F
                colSz[c] = n.layoutBounds.width;                            F
            if(n.layoutBounds.height > rowSz[r])                            F
```

```
                rowSz[r] = n.layoutBounds.height;                          F
            }                                                              F

            var x:Number = 0;
            var y:Number = 0;
            for(i in [0..<sz])
            {   def c:Integer = (i mod columns);
                def r:Integer = (i / columns).intValue();
                def n:Node = content[i];

                n.impl_layoutX = x;                                        G
                n.impl_layoutY = y;                                        G

                n.impl_layoutX += if(centerX)                              H
                    (colSz[c] - n.layoutBounds.width)/2                    H
                    else 0;                                                H
                n.impl_layoutY += if(centerY)                              H
                    (rowSz[r] – n.layoutBounds.height)/2                   H
                    else 0;                                                H

                if(c < (columns-1))                                        I
                {   x+=(colSz[c] + horizontalGap);                         I
                }                                                          I
                else                                                       I
                {   x=0;  y+=(rowSz[r] + verticalGap);                     I
                }                                                          I
            }
        }
    }
```

A Width in columns
B Center align?
C Gap between nodes
D Redo layout on reassignment
E Perform initial layout
F Find maximum col/row size
G Position node
H Center node, if required
I Next position

Listing 6.4 is based on the source code of the HBox and VBox layout nodes. It extends `Group`, and performs its layout by calling functions and variables inherited from its parent. The function type `impl_layout` is assigned to the code performing our layout, while calling the inherited function `impl_requestLayout()` actually causes the layout to occur (and said code to run).

The class has various initialization variables, these being:

- The `columns` variable determines how many columns the grid should have. The number of rows is calculated based upon this value and the size of the `content` sequence.
- The `centerX` and `centerY` variables switch on row or column center aligning (true is on).
- The `horizontalGap` and `verticalGap` variables control the gap between rows and columns.

We use the `init` block to assign our layout function, then we invoke an initial layout. To make the code simpler all the configuration variables are `public-init`, so they cannot be modifier externally once the object is created. It would actually be more useful if at least some of these variables could be manipulated after the node had been created; to make that possible `columns`, `centerX`, `centerY`, etc. should have '`on replace`' blocks attached to them which re-invoke the layout whenever they are reassigned. As an example of this, I've overridden `content` in the parent `Group` class to do just that.

But what of the layout code itself? In our `doGridBoxLayout()` function we first scan the `content` sequence to work out the width of each column and the height of each row. Then we do a second pass over the sequence, and use each node's `impl_layoutX` and `impl_layoutY` to set their position. These are internal node variables, used during layout to determine the position of a node. (Judging by their source code, it seems `HBox` and `VBox` favor these internal variables for node positioning over the documented `translateX` and `translateY`).

So there we have a functioning grid layout node class. Now all we need is some nodes to use it with—a problem we'll remedy next.

6.1.4 The Player class, version 1

We have our two custom classes, one a widget node, the other a layout node. Before going any further we should give them a trial run in a prototype version of our video player. Listing 6.5 will do this for us.

Listing 6.5 Player.fx (version 1)

```
package jfxia.chapter6;

import javafx.ext.swing.SwingSlider;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.scene.text.TextOrigin;
import javafx.stage.Stage;

def font = Font { name: "Helvetica";  size: 16; };                    A

Stage
{   scene: Scene
    {   content:
        [   Button                                                    B
            {   iconFilename: "arrow_l.png";
                action: function(ev:MouseEvent)
                {   println("Back");
                };
            } ,
            Button                                                    C
            {   translateX: 80;
                iconFilename: "arrow_r.png";
```

```
                action: function(ev:MouseEvent)
                {   println("Fore");
                };
            } ,

            GridBox                                                    D
            {   translateX: 185;  translateY: 20;
                columns: 3;
                centerY: true;
                horizontalGap: 10;  verticalGap: 5;

                var max:Integer=100;
                content: for(l in ["High","Medium","Low"])             E
                {   var sl:SwingSlider;
                    var contentArr =
                    [   Text                                           F
                        {   content: l;                                F
                            font: font;                                F
                            fill: Color.WHITE;                         F
                            textOrigin: TextOrigin.TOP;                F
                        } ,                                            F
                        sl = SwingSlider                               G
                        {   width: 200;                                G
                            maximum: max;                              G
                            value: max/2;                              G
                        } ,                                            G
                        Text                                           H
                        {   content: bind sl.value                     H
                                .intValue().toString();                H
                            font: font;                                H
                            fill: Color.WHITE;                         H
                            textOrigin: TextOrigin.TOP;                H
                        }                                              H
                    ];
                    max+=100;
                    contentArr;                                        I
                }
            }
        ];
        fill: Color.BLACK;
    };
    width: 550;  height: 140;
    title: "Player v1";
    resizable: false;
}
```

A Handy font declaration
B Left button
C Right button (note the translateX)
D Our GridBox in action
E Loop to add rows
F Left hand label
G The slider itself
H Bound display value
I Add row to GridBox

The code displays two buttons and some Swing sliders, using the two classes we developed previously.

I mentioned very briefly at the end of the section dealing with the `Button` class that our button code has a slight limitation, which we'd discuss later. Now is the time to reveal all. The click animation used in our `Button` class introduces slight headache: the animation effect expands beyond the size of the button itself. Although creating a *cool* zoom visual, it means padding is required around the perimeter, accommodating the effect when it occurs. This is the purpose of the transparent `Rectangle` which sits below all other nodes in the `Button`'s internal scene graph.

Without this padding the button would grow in size as the animation played, which might cause its parent layout node to continually re-evaluate its children, resulting in a jostling effect on screen as other nodes moved to accommodate the button.

To solve this problem we need to absolutely position our buttons, preferably overlapped to mask their over-sized padding. And this is exactly what listing 6.5 does, by using the `translateX` variable.

Following the two buttons in the listing we find an example of our `GridBox` in use. Its content is formed using a `for` loop, which adds three nodes (one whole row) with each pass. The first and last are `Text` nodes, while the middle is a `SwingSlider` node.

The `javafx.scene.text.Text` nodes simply display a string using a font, not unlike the `SwingLabel` class. Strictly speaking they are more like JavaFX shape nodes (`Rectangle`, `Circle`, etc.) than UI widgets, however, so have a concept of a `fill` (body color) and a `stroke` (perimeter color), as well as other shape-like capabilities. By default a `Text` node's co-ordinate origin is measured from the font's baseline (the imaginary line on which the text rests, like the ruled lines on writing paper), but in our listing we re-locate the origin to the more convenient top left corner of the node.

The `SwingSlider` is, as its name suggests, a wrapper around Swing's `JSlider` component. It allows the user to pick a value between a given minimum and a maximum by dragging a thumb along a track. The final `Text` node on each row is bound to the current value of this slider, and its text content will change when the associated slider is adjusted.

Version one of our application is complete!

> **WARNING: SWING WILL SOON BE SLUNG**
>
> Be warned: the Swing library is being pushed firmly into the background once JavaFX gets its own controls API. The Swing slider in this project is only here in lieu of the `javafx.scene.control` package getting its own slider control, which may already have happened by the time you read this chapter.

6.1.5 Running version 1

Running version one gives us a basic control panel, as revealed by figure 6.6 below. Although the code is compact, the results are hardly crude. The buttons are fully functional, have their own event handler into which code can be plugged, and sport a really cool animation effect when pressed. The layout node makes building the slider UI much easier, yet still appropriately configurable.

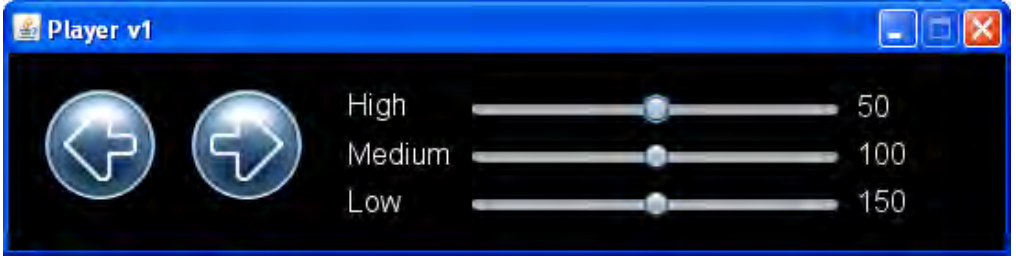


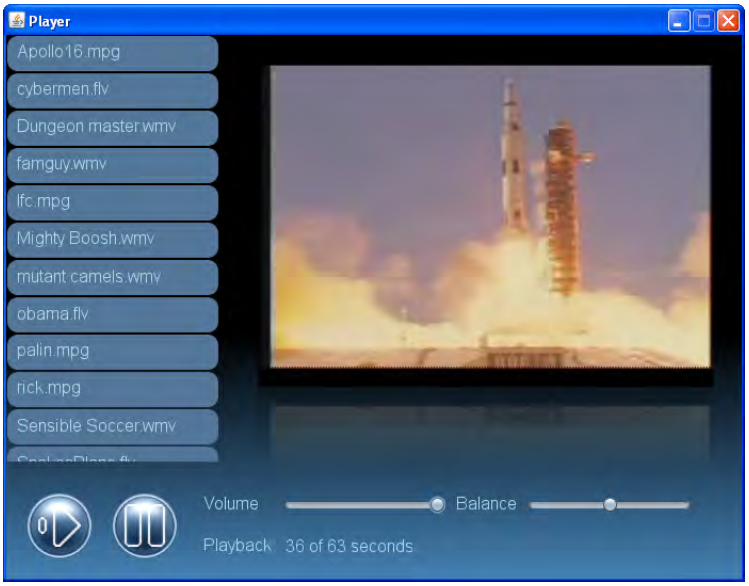
Figure 6.6 Our custom button and layout nodes, on display.

I'll leave it up to you, the reader, to polish off our custom classes with your own bells and whistles. The `GridBox` in particular could become a really powerful layout class with not a great deal of extra work. The additional code wouldn't be of value from a demonstration view point (that's why I didn't add it myself), but I encourage you to use version one as a test bed to try out your own enhancements.

So much for custom widgets, did I hear someone ask when will we start playing with video? Good question. In the second, and final, part of this project we develop our most ambitious custom node yet—and, yes, finally we get to play with some video. (Woo-hoo!)

6.2 The Video Player, version 2

In this part of the chapter we have two objectives. The first is to incorporate a video playback node into our scene graph; the second is to develop a custom node for listing and



choosing videos. Figure 6.7 shows what we're after.

Figure 6.7 Lift off! Our control panel (bottom) is combined with a new list (left hand side) and video node (central) to create the final player.

Figure 6.7 shows the two new elements in action. The list allows the user to pick a video, and the video playback node will show it. Our control panel will interact with the video as it plays, pausing and restarting the action, and adjusting the sound. The list widget down the side will use tween based animations, not only to control rollover effects, but also its scrolling.

We need to develop this list widget first, so that's where we'll head next.

6.2.1 The List class: a complex multi-part widget

The `List`/`ListPane` widget is quite complex, indeed so complex that it's been broken into two classes. `List` is an interior node for displaying a list of strings and firing `action` events when they are clicked. `ListPane` is an outer container node which allows the `List` to be scrolled. Below, in figure 6.8, we can see how the list parts slot together.

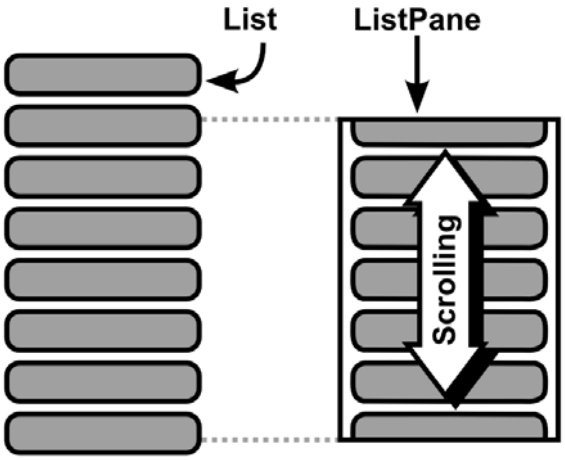


Figure 6.8 The List and ListPane classes will allow us to present a selection of movie files for the user to pick from.

Rather than using a scrollbar, I thought we'd attempt something a little different—the list will work in a vaguely *iPhone* like fashion. A press and drag will scroll the list, with inertia when we let go, while a quick press and release will be treated as a click on a list item. We start with just the inner `List` node, which I've broken into two parts to avoid page flipping. The first part is listing 6.6.

Listing 6.6 List.fx (part 1)

```
package jfxia.chapter6;

import javafx.animation.Interpolator;
```

```
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.scene.text.TextOrigin;

package class List extends CustomNode
{   package var cellWidth:Number = 150;                                    A
    package var cellHeight:Number = 35;                                    A

    public-init var content:String[];
    public-init var font:Font = Font {};
    public-init var foreground:Color = Color.LIGHTBLUE;
    public-init var background:Color = Color.web("#557799");
    public-init var backgroundHover:Color = Color.web("#0044AA");
    public-init var backgroundPressed:Color = Color.web("#003377");
    public-init var action:function(n:Integer);

    def border:Number = 1.0;
    var totalHeight:Number;

    override function create() : Node                                      B
    {   VBox { content: build(); }
    }
    // ** Part two below
```

A List item dimensions
B Create scene graph

At the head of the code we have our usual collection of variables for controlling the class:

- The `cellWidth` and `cellHeight` are the dimensions of the items on screen. They need to be manipulated by the `ListPane` class, so we've given them package visibility.
- The `content` variable holds the strings which define the list labels.
- The variables `font`, `foreground`, `background`, `backgroundHover` and `backgroundPressed` control the font and colors of the list.
- The function type `action` is our callback function.
- There are also two *private* variables, `border` holds the gap around items in the list, and `totalHeight` stores the pixel height of the list.

Looking at the `create()` code we see a `VBox` being fed content by a function called `build()`. `VBox` stacks its contents one underneath the other—precisely the functionality we need. But what about the `build()` function which creates its contents? Take a look at the next part of the code, in listing 6.7.

Listing 6.7 List.fx (part 2)

```
// ** Part one above
function build() : Node[]
{   for(i in [0..<sizeof content])                                          C
    {   var t:Text;
        var r:Rectangle;
        def g = Group
        {   content:
            [   Rectangle                                                   D
                {   width: bind cellWidth;
                    height: bind cellHeight;
                    opacity: 0.0;
                },
                r = Rectangle                                               E
                {   x:border;  y:border;
                    width: bind cellWidth-border*2;
                    height: bind cellHeight-border*2;
                    arcWidth:20;  arcHeight:20;
                    fill: background;
                    onMouseExited: function(ev:MouseEvent)
                    {    anim(r,background);
                    };
                    onMouseEntered: function(ev:MouseEvent)
                    {    r.fill = backgroundHover;
                    }
                    onMousePressed: function(ev:MouseEvent)
                    {    r.fill = backgroundPressed;
                    }
                    onMouseClicked: function(ev:MouseEvent)
                    {    r.fill = backgroundHover;
                        if(action!=null) { action(i); }
                    }
                } ,
                t = Text                                                    F
                {   x: 10;  y: border;
                    content: bind content[i];
                    font: bind font;
                    fill: bind foreground;
                    textOrigin: TextOrigin.TOP;
                }
            ];
        };
        t.y += (r.layoutBounds.height-                                      G
            t.layoutBounds.height)/2;                                       G
        totalHeight += g.layoutBounds.height;
        g;
    }
}

function anim(r:Rectangle,c:Color) : Void                                   H
{   Timeline
    {   keyFrames:
        [   KeyFrame
            {   time: 0.5s;
                values:
                [   r.fill => c
```

```
                                tween Interpolator.LINEAR
                        ];
                    }
                ]
            }.playFromStart();
        }
    }
```

C For each list item
D Hidden sizing rectangle
E List rectangle
F Label text
G Center vertically
H Animate background

The `build()` function returns a sequence of nodes, each a `Group` consisting of two `Rectangle` nodes and a `Text` node. The first node enforces an empty border on all four sides of each item. The second `Rectangle` is the visible box for our item, it also houses all the mouse event logic. Finally, we have the label itself, as a `Text` node. For easy handling we use the top of the text as its co-ordinate origin, rather than its baseline.

Both the background `Rectangle` and `Text` are assigned to variables (as JavaFX Script is an expression language this won't prevent them from being added to the `Group`). Why? Take a look at the code immediately after the `Group` declaration–you'll see, using the variables, we vertically center the `Text` within the `Rectangle`. That's why we needed references to them.

Now let's consider the mouse handlers. Entering the item sets the background rectangle fill color to `backgroundHover`, while exiting the item kicks off a `Timeline` (via the `anim()` function) to slowly return it to `background`. This creates a trail effect as the mouse moves over the list.

Pressing the mouse button sets the color to `backgroundPressed`, but we don't bother with the button release event; instead we look for the higher level *clicked* event, created when the user stabs the button, as opposed to a press and hold. The click event fires off our own `action` function, which can be assigned by outside code to respond to list selections.

But the `List` class is only half of the widget, it's almost useless without its sibling, the `ListPane` class.

6.2.2 The ListPane class: scrolling and clipping a scene graph

Now that we've seen the `List` node, let's consider the outer container which scrolls it. Check out listing 6.8, up next.

Listing 6.8 ListPane.fx (part 1)

```
package jfxia.chapter6;

import javafx.animation.Interpolator;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.Node;
```

```
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;

package class ListPane extends CustomNode
{   public-init var content:List;

    package var width:Number = 150.0                                    A
        on replace oldVal = newVal                                      A
        {   if(content!=null)                                           A
                content.cellWidth = newVal;                             A
        };                                                              A
    package var height:Number = 300.0;
    package var scrollY:Number = 0.0                                    B
        on replace oldVal = newVal                                      B
        {   if(content!=null)                                           B
                content.translateY = 0-newVal;                          B
        };                                                              B

    var clickY:Number;                                                  C
    var scrollOrigin:Number;                                            C
    var buttonDown:Boolean = false;                                     C
    var dragDelta:Number;                                               D
    var dragTimeline:Timeline;                                          D
    var noScroll:Boolean =
        bind content.layoutBounds.height < this.height;
    // ** Part two below
```

A Pass width on to List
B Position List within pane
C Drag variables
D Inertia animation variables

The above code is the first part of our `ListPane` class, housing the `List` we created earlier. The exposed variables are quite straight forward:

- The `content` is our `List`.
- The `width` and `height` are the dimensions of the node. The `width` is passed on to the `List`, where it's used to size the list items.
- The `scrollY` variable is the scroll position of the `List` within our pane. The value is actually the `List` position relative to the `ListPane`, which is why it's negative. To scroll to pixel position 40, for example, we position the `List` at -40 compared to its container pane.

The private variables control the drag and the animation effect:

- To move the `List` we need to know how far we've dragged the mouse during this *operation*, and where the `List` was before we started to drag. The private variable `clickY` records where inside the pane the mouse was when its button was pressed down, and `scrollOrigin` records its scroll position at that time. Meanwhile `buttonDown` is a handy flag, recording whether we're in the middle of a drag operation or not.

- In order to create the inertia effect we must know how fast the mouse is traveling before its button was released, and `dragDelta` records that for us. We also need a `Timeline` for the effect, hence `dragTimeline`.
- If the `List` is smaller than the `ListPane` we want to disable any scrolling or animation. The flag `noScroll` is used for this very purpose.

So much for the class variables. What about the actual scene graph and mouse event handlers? For those we need to look at listing 6.9, next.

Listing 6.9 ListPane.fx (part 2)

```
// ** Part one above
override function create() : Node
{   Group
    {   content:
        [   this.content ,                                              E
            Rectangle                                                   F
            {   width: bind this.width;
                height: bind this.height;
                opacity: 0.0;
                onMousePressed: function(ev:MouseEvent)
                {   animStop();
                    clickY = ev.y;
                    scrollOrigin = scrollY;
                    buttonDown = true;
                };
                onMouseDragged: function(ev:MouseEvent)
                {   def prevY = scrollY;
                    updateY(ev.y);
                    dragDelta = scrollY-prevY;
                };
                onMouseReleased: function(ev:MouseEvent)
                {   updateY(ev.y);
                    animStart(dragDelta);
                    dragDelta = 0;
                    buttonDown = false;
                };
                onMouseWheelMoved: function(ev:MouseEvent)
                {   if(buttonDown == false)
                    {   scrollY = restrainY
                        (   scrollY + ev.wheelRotation
                                * content.cellWidth
                        );
                    }
                };
            }
        ];
        clip: Rectangle                                                 G
        {   x:0;  y:0;                                                  G
            width: bind this.width;                                     G
            height: bind this.height;                                   G
        }                                                               G
    }
}
```

```
        function updateY(y:Number) : Void
        {   if(noScroll) { return; }
            scrollY = restrainY( scrollOrigin-(y-clickY) );
        }
        function restrainY(y:Number) : Number                                   H
        {   def h = content.layoutBounds.height-height;
            return
                if(y<0)  0
                else if(y>h)  h
                else  y;
        }

        function animStart(delta:Number) : Void
        {   if(dragDelta>5 and dragDelta<-5) { return; }
            if(noScroll) { return; }

            def endY = restrainY(scrollY+delta*15);
            dragTimeline = Timeline                                             I
            {   keyFrames:                                                      I
                [   KeyFrame                                                    I
                    {   time: 1s;                                               I
                        values:                                                 I
                        [   scrollY => endY                                     I
                                tween Interpolator.EASEOUT                      I
                        ];                                                      I
                    }                                                           I
                ]                                                               I
            };                                                                  I
            dragTimeline.playFromStart();
        }
        function animStop() : Void
        {   if(dragTimeline!=null)
            {    dragTimeline.stop();
            }
        }
    }
```

E List node
F Background and mouse events
G Constrain visible area
H Limit scroll to list size
I Inertia time line

The scene graph for `ListPane` consists of two nodes: the `List` itself and a `Rectangle` which handles our mouse events.

- When `onMousePressed` is triggered we stop any inertia animation which may already be running, store the initial mouse y co-ordinate and the current list scroll position, then flag the beginning of a drag operation.
- When `onMouseDragged` is called we update the `List` scroll position and store the number of pixels we moved this update (used to calculate the speed of the inertia when we let go). The `restrain()` function prevents the `List` from being scrolled off its top or bottom.

- Finally, when the `onMouseReleased()` function is called, it updates the `List` position, kicks off the inertia animation, and resets the `dragDelta` and `buttonDown` variables ready for next time.
- There's also a handler for the mouse scroll wheel, `onMouseWheelMoved()`, which should only work when we're not in the middle of a drag operation (we can drag, or *wheel*, but not both at the same time!)

You'll note the `Group` employs a `Rectangle` as a clipping area. Clipping areas are the way to show only a restricted view of a scene graph. Without this, the `List` nodes would spill outside the boundary of our `ListPane`. The clipping assignment creates the *view port* behavior our widget requires, as demonstrated earlier in figure 6.8.

Finally, let's consider the `animStart()` function, which kicks off the inertia animation. The `delta` parameter is the number of pixels the pointer moved in the mouse dragged event immediately before the button release. We use this to calculate how far the list will continue to travel. If the mouse movement was too slow (less than 5 pixels), or the `List` is too small to scroll, we just exit. Otherwise a `Timeline` animation is set up and started.

The list widget was our most ambitious piece of scene graph code yet. The end result, complete with hover effect as the mouse moves over the list, is shown in figure 6.9. Even though it supports a lavish smooth scroll, and animated reactions to the mouse pointer, it didn't take much more than a couple of hundred lines of code to write. It just shows how easy it is to create impressive UI code in JavaFX.

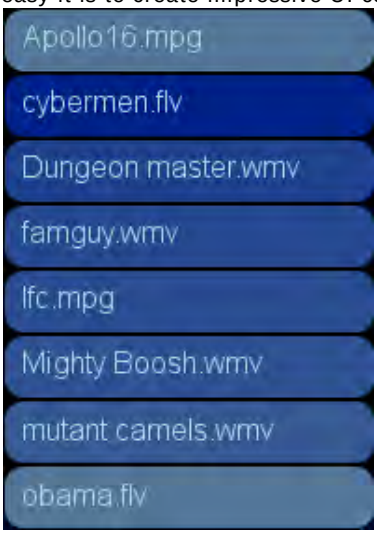


Figure 6.9 A closer look at our List and ListPane widget, with hover effect visible on the background of the list items.

In the next part we'll delve into the exciting world of multimedia, as we plug our new list widget into the project application, and use it to trigger video playback.

6.2.3 Using media in JavaFX

The time has come to learn how JavaFX handles media, such as the video files we'll be playing in our application. Before we look at the JavaFX Script code itself, let's invest a little time in learning about the theory. We'll start with figure 6.10.

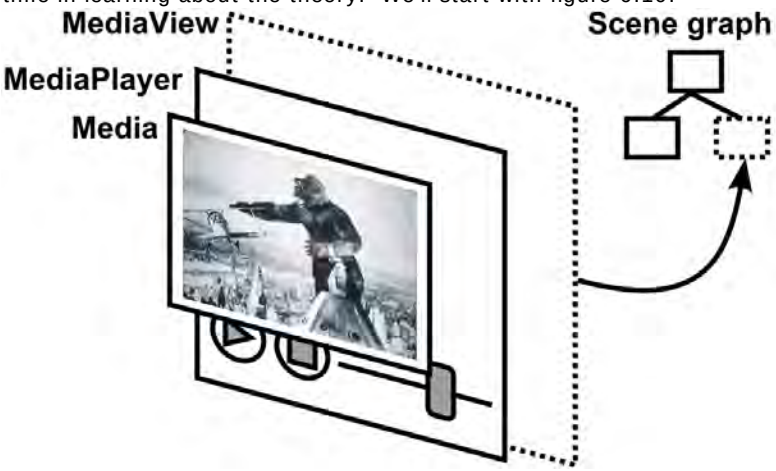


Figure 6.10 Like other JavaFX user interface elements, video is played via a dedicated MediaView scene graph node. (Note: MediaPlayer is not a visual element, the control icons are symbolic.)

To plug a video into the JavaFX scene graph takes three classes, located in the `javafx.scene.media` package. They are demonstrated in figure 6.10; starting from the *outside*, and working in, they are:

- The `MediaView` class, which acts as a bridge between the scene graph and any visual media which needs displaying within it.
- The `MediaPlayer` class, which controls how the media is played. For example: stopping, restarting, skipping forward or backwards, slowed down or sped up. Important: `MediaPlayer` merely permits programmatic control of media, it provides no actual user interface controls (figure 6.10 is symbolic).
- The `Media` class, which encapsulates the actual video and/or audio data to be played by the `MediaPlayer`.

As with images, JavaFX prefers to work with URLs rather than directly with local directory paths and file names. If you read the API documentation you'll see the classes are designed to work with different types of media, and make allowances for data being streamed across a network.

The data formats supported fall into two categories. JavaFX will make use of the runtime operating system's media support, allowing it to play formats supported on the current

platform. For cross platform applications, however, JavaFX also includes its own codec, which can be trusted to run no matter what the capabilities of the underlying operating system.

Table 6.1 JavaFX media support on various operating systems

Platform	Codecs	Formats
Mac OS X 10.4 and above (Core Video)	**Video:** H.261, H.263, and H.264 codecs. MPEG-1, MPEG-2, and MPEG-4 Video file formats and associated codecs (such as AVC). Sorenson Video 2 and 3 codecs. **Audio:** AIFF, MP3, WAV,MPEG-4 AAC Audio (.m4a, .m4b, .m4p), MIDI.	3GPP / 3GPP2, AVI, MOV, MP4, MP3.
Windows XP/Vista (DirectShow)	**Video:** Windows Media Video,H264 (as an update) **Audio:** MPEG-1, MP3, Windows Media Audio, MIDI.	MP3, WAV, WMV, AVI, ASF.
JavaFX (cross platform)	**Video:** On2 VP6. **Audio:** MP3.	FLV, FXM (Sun defined FLV subset), MP3.

Table 6.1 shows the various support on different platforms. At the time of writing the details for Linux media support were not available, although the same mix of native and cross platform codecs is expected.

The cross platform video comes from a partnership deal Sun made with On2 for their VidoeVP6 decoder. On2 are best known for providing the software supporting Flash's own video decoder. The VP6 decoder plays FXM media on all JavaFX platforms, including mobile (and presumably TV too) when it arrives, without any extra software installation. Regrettably the only encoder for the On2 format at the time of writing seems to be *On2 Flix*, a proprietary commercial product.

Now we understand the theory, let's push on to the final part of the project, where we build a working video player.

6.2.4 The Player class, version 2: video and linear gradients

We now have all the pieces, it only remains to pull them together. The listing which follows is our largest single source file yet, almost two hundred lines (just be thankful this isn't a Java book, or it could have been ten times that). To avoid the dreaded page flip, I've broken it up into three parts, each dealing with different stages of the application. The opening part is listing 6.10.

Listing 6.10 Player.fx (version 2, part 1)

```
package jfxia.chapter6;

import javafx.ext.swing.SwingSlider;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.Reflection;
import javafx.scene.input.MouseEvent;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaView;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Stop;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.scene.text.TextOrigin;
import javafx.stage.Stage;

import java.io.File;
import javax.swing.JFileChooser;


var sourceDir:File;
var sourceFiles:String[];

def fileDialog = new JFileChooser();                                    A
fileDialog.setFileSelectionMode(                                        A
   JFileChooser.DIRECTORIES_ONLY);                                      A
def ret = fileDialog.showOpenDialog(null);                              A
if(ret == JFileChooser.APPROVE_OPTION)
{   sourceDir = fileDialog.getSelectedFile();                           B
    if(sourceDir.isDirectory() == false)                                B
    {   println("{sourceDir} is not a directory");                      B
        FX.exit();                                                      B
    }                                                                   B
    def files:File[] = sourceDir.listFiles();                           C
    for(i in [0 ..< sizeof files])                                      C
    {   def fn:String = files[i].getName().toLowerCase();               C
        if(fn.endsWith(".mpg") or fn.endsWith(".mpeg")                  C
            or fn.endsWith(".wmv") or fn.endsWith(".flv"))              C
        {   insert files[i].getName() into sourceFiles;                 C
        }
    }

}
else
{   FX.exit();
}
// ** Part two below
```

A Select a directory
B Check valid selection
C Create video file list

When run, the program asks for a directory containing video files using Swing's own `JFileChooser` class. This time we're not using JavaFX wrappers around a Swing

component, we're actually creating and using the raw Java class itself. Having created the chooser we tell it to only list directories, then show it and wait for it to return. Assuming the user selected a directory, we run through all its files looking for potential videos based on their filename extension, populating the `sourceFiles` sequence when found.

Assuming we continue running past this piece of code, the next step (listing 6.11) is to set up the scene graph for our video player.

Listing 6.11 Player.fx (version 2, part 2)

```
// ** Part two above
def margin = 10.0;
def videoWidth = 480.0;                                                    D
def videoHeight = 320.0;                                                   D
def reflectSize = 0.25;
def font = Font { name: "Helvetica";  size: 16; };

var volumeSlider:SwingSlider;                                              E
var balanceSlider:SwingSlider;                                             E

def list:ListPane = ListPane                                               F
{   content: List
    {   content: sourceFiles;
        font: font;
        action: function(i:Integer)                                        G
        {   mPlayer.media = Media                                          G
            {   source: getVideoPath(i);                                   G
            }                                                              G
            mPlayer.play();                                                G
        };                                                                 G
    };
    width: 200;
    height: bind mView.layoutBounds.height;                                H
}

var mPlayer:MediaPlayer;                                                   I
def mView:MediaView = MediaView
{   def tWidth = videoWidth + margin*2;
    def tHeight = videoHeight*(1.0+reflectSize) +  margin*2;

    translateX: bind list.layoutBounds.width +                             J
        (tWidth-mView.layoutBounds.width)/2                                J
    translateY: bind                                                       J
        (tHeight-margin-mView.layoutBounds.height);                        J
    fitWidth: videoWidth;
    fitHeight: videoHeight;
    preserveRatio: true;
    effect: Reflection                                                     K
    {   fraction: reflectSize;                                             K
        topOpacity: 0.25;                                                  K
        bottomOpacity: 0.0;                                                K
    };                                                                     K
    mediaPlayer: mPlayer = MediaPlayer                                     L
    {   volume:  bind volumeSlider.value / 100.0;                          L
        balance: bind balanceSlider.value / 100.0;                         L
        onEndOfMedia: function()                                           L
```

```
        {   mPlayer.currentTime = 0s;                                   L
        }                                                               L
    }                                                                   L
};
def vidPos = bind mPlayer.currentTime.toSeconds() as Integer;           M

def panel:Group = Group                                                 N
{   translateY: bind mView.layoutBounds.height + margin;
    content:
    [   Button                                                          O
        {   iconFilename: "play.png";
            action: function(ev:MouseEvent)
            {   mPlayer.play();
            }
        } ,
        Button                                                          P
        {   translateX: 80;
            iconFilename: "pause.png";
            action: function(ev:MouseEvent)
            {   mPlayer.pause();
            };
        } ,
        GridBox                                                         Q
        {   translateX:185;  translateY:20;
            columns:4;
            centerY: true;
            horizontalGap:10;  verticalGap:20;
            content:
            [   makeLabel("Volume") ,                                   R
                volumeSlider = SwingSlider                              R
                {   width: 150;                                         R
                    value: 100;                                         R
                    maximum: 100;                                       R
                } ,                                                     R

                makeLabel("Balance") ,                                  S
                balanceSlider = SwingSlider                             S
                {   width: 150;                                         S
                    value: 0;                                           S
                    minimum: -100;  maximum: 100;                       S
                } ,                                                     S

                makeLabel("Playback") ,                                 T
                Text                                                    T
                {   content: bind                                       T
                        if(mPlayer.media!=null)                         T
                            "{vidPos} seconds"                          T
                        else                                            T
                            "No video";                                 T
                    font: font;                                         T
                    fill: Color.LIGHTBLUE;                              T
                    textOrigin: TextOrigin.TOP;                         T
                }                                                       T
            ];
        }
    ];
};
```

```
// ** Part three below
```
D Video display dimensions
E Volume/balance sliders
F List widget
G Action: create then play media
H Height matches MediaView
I The MediaView node
J Always rests on area baseline
K Reflection under video
L MediaPlayer inside MediaView
M Video position/duration (handy)
N Control panel
O Play button
P Pause button
Q GridBox layout
R Volume slider
S Balance slider
T Postion display

Above we create the various parts of our scene graph, which will be plugged into the application's `Stage` in part three (below). There are three main parts: `list` is the scrolling list of videos from which the user can select, `mView` is the video display area itself, and `panel` is the control panel which runs along the bottom of the window. You'll notice the `MediaPlayer` is also given its own variable, `mPlayer`.

The list widget is constructed from the two classes, `List` and `ListPane`, we developed earlier. Its contents are the filenames from the directory returned by `JFileChooser`. The `action` event takes the selected list index and turns it into a URL (thanks to the `getVideoPath()` function we'll see later), creates a new `Media` object from it, plugs the `Media` object into `mPlayer`, then starts it playing.

The video area, `mView`, has a lot of code at its start to work out sizing and positioning. Figure 6.11 shows relationship of the key variables.

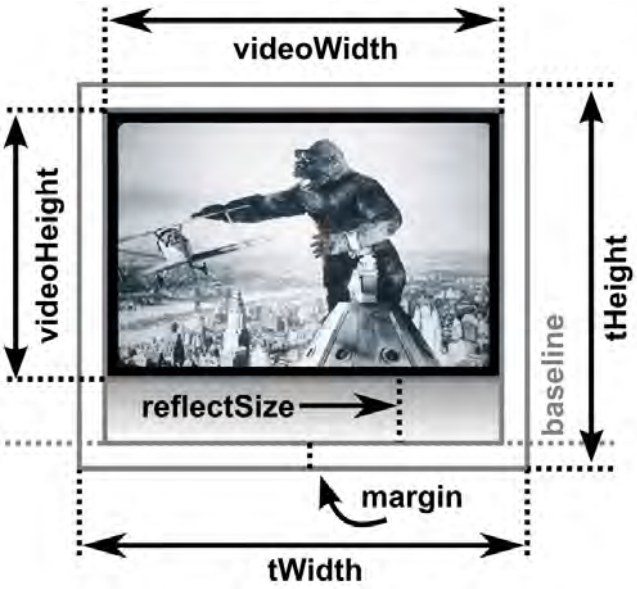


Figure 6.11 The video area layout and sizing is controlled by various variables, some at the script level and others local to the scene graph node itself.

The script level variables `videoWidth` and `videoHeight` determine the pixel size of the actual video node, `reflectSize` is the proportion of the node which gets reflected below it (see figures 6.1 and 6.7), and `margin` is the border around the whole video and reflection. The local variables tWidth and tHeight are the total dimensions of the node, taking into account video size, reflection effect and margin.

The `fitWidth` and `fitHeight` parameters are set on the `MediaView` node, causing any video to be resized to fit those dimensions (`preserveRatio` set to true ensures the video is never stretched out of proportion). We want our video to appear to rest on a reflective surface, so the vertical position (`translateY`) of the video/reflection is measured from the baseline of the area–in other words the vertical alignment is 'bottom'.

The reflection effect may look impressive, but in JavaFX applying any visual effect to a section of the scene graph is as easy as assigning the given node's `effect` variable. If you look in the API documentation for the `javafx.scene.effect` package you'll see all manner of different effects you can apply; we'll be looking at more of them in future chapters. The `Reflection` adds a mirror below its node, with a given top/bottom opacity.

The media player node itself is quite simple. Its `volume` and `balance` variables are bound to sliders in the control panel, and it has an event handler (`onEndOfMedia`) which rewinds the video back to the start once it reaches the end.

Finally, the control panel, aka the `panel` variable, will be familiar from version one of the project. The only substantial difference is now the sliders are plugged into actual video

player variables. The makeLabel() function is simple a convenience for creating label text; it appears in part three of the code. And talking of part three, here's listing 6.12:

Listing 6.12 Player.fx (version 2, part 3)

```
Stage
{   scene: Scene
    {   content:                                                    U
        [   list,mView,panel                                        U
        ];                                                          U
        fill: LinearGradient                                        V
        {   endX:0;  endY:1;                                        V
            proportional: true;                                     V
            stops:                                                  V
            [   Stop                                                V
                {   offset:0.55;  color:Color.BLACK;                V
                } ,                                                 V
                Stop                                                V
                {   offset:1;  color:Color.STEELBLUE;               V
                }                                                   V
            ];                                                      V
        };                                                          V
    };
    title: "Player v2";
    resizable: false;
}



function makeLabel(str:String) : Text                               W
{    Text
    {   content: str;
        font: font;
        fill: Color.LIGHTBLUE;
        textOrigin: TextOrigin.TOP;
    };
}

function getVideoPath(i:Integer) : String                           X
{   def f = new File(sourceDir,sourceFiles[i]);
    return f.toURI().toString();
}
```

U Scene graph bits
V Linear gradient background
W Handy label maker function
X Convert list filename to URL

The final part of our application. Whew!

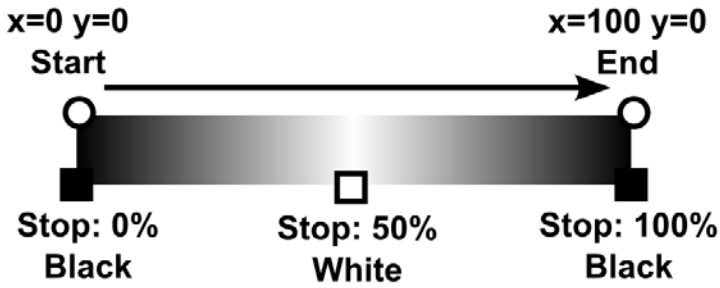
We see here the three nodes we created in the second part (`list`, `mView` and `panel`) are hooked into our scene. At the bottom of the listing we see the `makeLabel()` and `getVideoPath()` convenience functions we used previously. But what's that in the middle of the listing, plugged into the scene's `fill` parameter?

That is a `LinearGradient`, and it is responsible for the graduated fill color which sits behind the whole window's contents. If you think it looks rather odd, don't worry–I've devoted an entire section to its unlocking it secrets, next.

6.2.5 Creating varying color fills with LinearGradient

Instead of using a boring flat color for the window background, in listing 6.12 (above) we create a `LinearGradient` and use it as the scene's fill. We can do this because the `Scene` class accepts a `javafx.scene.paint.Paint` instance as its background fill. The `Paint` class is a base for any object which can to used to determine how the pixels in a shape will be drawn—the flat colors we used in previous examples were also types of `Paint`, albeit not very exciting ones.

A gradient paint is one which transitions between a set of colors as it draws a shape. Good examples of linear gradients might include a color spectrum or a chrome metal effect,



like figure 6.12.

Figure 6.12 A gradient paint is one where the pixel tone changes over the course of a given area.

Think about how the color gradually changes across the painted area. To define a linear gradient we need two things: a line with a start and end point (a path to follow), and a list of colors on said line plus where they are positioned. For a simple rainbow spectrum we might define a horizontal (or vertical) line, with each color *stop* spaced equally along its length.

The line can function either as a relative scale, or as absolute pixel coordinates. But what's the difference? Take a look at the examples in figure 6.13.

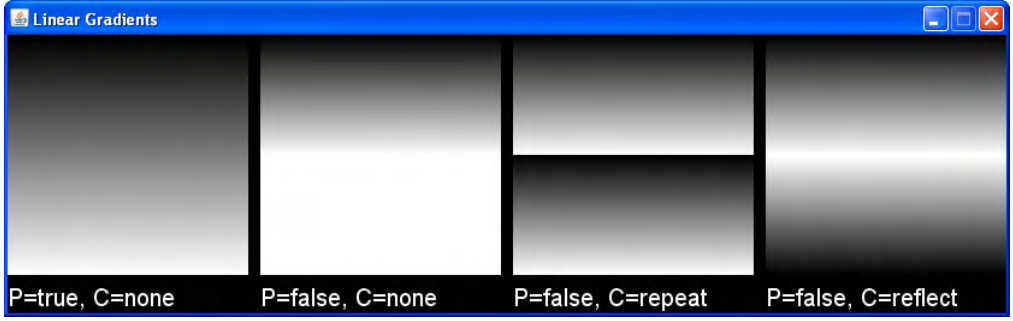


Figure 6.13 A proportional (P) gradient scaled to full height. Then three non-proportional examples, gradient (0,0) to (0,100), painted onto 200 x 200 sized rectangles with various cycle (C) methods.

When `proportional` is set the line co-ordinates, (`startX`, `startY`, `endX` and `endY`) are scaled across a virtual co-ordinate space from 0 to 1, which is stretched to fit the actual painted area whenever the paint is applied. Without `proportional` set the line start and end co-ordinates are absolute pixel positions. This means if you define a vertical gradient line of (0,0) to (0,100), but then paint an area of 200x200 pixels, the transition will not cover the entire painted area.

By setting an parameter called `cycleMethod` we can control how the remainder will paint. The default option extends the color at either end of the gradient line. Alternatively we can repeat the gradient, or reflect it by painting it backwards.

Incidentally, `LinearGradient` creates stripes of color along a gradient line, but you might also want to check out its cousin, `RadialGradient`, which paints circular patterns. It can be particularly useful for creating pleasing 3D ball effects.

6.2.6 Running version 2

Running version two of the project gives us our video player, as shown below. Selecting a file from the list on the left will play it in the central area, complete with snazzy reflection over the shaded *floor*.

Figure 6.14 shows what you should expect when firing up the application, selecting a directory, and playing a video (especially if you're a Mighty Reds fan).

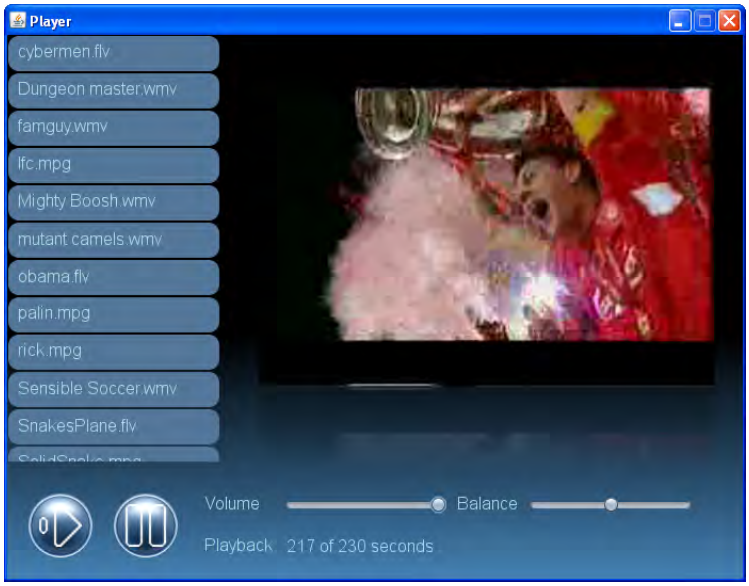


Figure 6.14 You'll never walk alone: re-live favorite moments with your own home-made video player (like your soccer team lifting the “Champion's League” trophy.)

Okay, so perhaps it's not the most functional player in the world, but it's still very impressive when you consider how little work we actually needed to pull it off. Just think how long it would have taken to code all the UI and effects using Java, or C++. And that's before we even *think* about getting video playback working.

6.3 Summary

In this chapter we've seen in greater depth how to use the standard scene graph nodes to build fairly sophisticated UI widgets, we've looked at how to include images and video into our JavaFX applications, and we've played around with gradient paints. We've also seen our first example of plugging an effect (reflection) into the scene graph.

Writing good scene graph code is all about planning—JavaFX gives you some powerful raw building blocks; you have to consider the best way to fit them together for the effect you're trying to achieve. Always be aware that a scene graph is a different breast to something like Swing. It's a structured representation of the graphics on screen, and as such we need to ensure it includes layout and spacing information, because we don't have direct control of the actually pixel painting link in Swing. Transparent shapes can be used to enforce spacing around parts of our scene graph, but they can also be used as a central event target.

Hopefully the source code in this chapter has given you ideas about how to write your own UI nodes. Feel free to experiment with the player, filling in its gaps and adding new features. Or take the widgets and develop them further in your own applications.

In the next chapter we're shifting focus from moving images to static ones, as we explore pulling data into our programs from web services. We'll also try applying a few more effects to our nodes, to make them look even more impressive.

But for now, enjoy the movie!

7

Web services with style

In this chapter we're going to cover a range of exciting JavaFX features, and possibly the most fun project thus far! Certainly, the largest in this book. In previous chapters we were still learning the ropes, so to speak, but now bind and triggers are becoming second nature, and the scene graph is no longer a strange alien beast, we can move on to some of the power tools the JavaFX API has to offer.

One such power tool is the ability to change the look of a user interface without having to recompile the source code. We can do this using stylesheets, but a UI component must be written to allow itself to be styled in such a way. Fortunately JavaFX comes ready-made with a mechanism to do this; this chapter will demonstrate how to write widgets which can exploit it.

We'll also be seeing how to call a web service, and parse its XML response. As more of our data is moving on-line, and hidden behind web services, knowing how to exploit their power from within our own software becomes crucial. This chapter will show just how it's done.

Finally, we look at a way to take the pain out of making animation effects. As if the animation tools built into JavaFX Script weren't enough, JavaFX also includes a whole library of off-the-shelf *transition* classes. When applied to scene graph nodes we can make them move, spin, scale and fade with ease.

This chapter has a lot to cover, but by the time you reach its end you'll have experienced practical examples of the majority of JavaFX techniques and libraries. Although there are still a few juicy morsels in the remaining chapters, after this we'll focus more on how to apply our newfound skills in different circumstances.

We might as well get started.

7.1 Our project: a Flickr image viewer

Everyone and his dog is writing demos to exploit the Flickr web service API. It's not hard to understand why. Located at `www.flickr.com`, the site is an on-line photo gallery, where

members of the public can upload and arrange their digital masterpieces for others to browse. It's highly addictive trawling though random photos, some looking clearly amateur, but others shockingly professional. Of course, as programmers we just want to know how cool the programming APIs are! And, as it happens, Flickr's API is pretty cool (hence everyone and his dog using it).

Given that there's no shortage of existing examples, you may perhaps be wondering why I decided to go with Flickr for this book? Firstly, it's a well known API with plenty of documentation and programmers who are familiar with it–important for anyone playing with the source code after the chapter has been read. Secondly, I wanted to show that photo gallery applications don't have to be boring (witness figure 7.1 below) particularly when you have a tool like JavaFX at your disposal.

Figure 7.1 Our photo viewer will allow us to contact the on-line photo service, view thumbnails from a gallery, then select and 'toss' a full sized image onto a 'desktop' as if it was a real photo.

The application we're going to build will run full screen. It will use a web service API to fetch details of a particular gallery, then show thumbnails in a strip along the bottom of the screen, one page at a time. Selecting a thumbnail will cause the image to spin onto the main desktop (the background) display, looking as if it's a printed photograph. The desktop can be dragged to move the photos, and as more pictures are 'dropped' onto it, older ones (at the bottom of the heap) gracefully fade away.

Thanks, Sally!

I'd like to thank Sally Lupton, who kindly allowed her gallery to be used to illustrate figures 7.1, 7.6 and 7.7. Her *Superlambanana* photos were certainly a lot nicer than anything your humble author could produce.

7.1.1 The Flickr web service

A web service is a means of communicating between two pieces of software, typically on different networked computers. The client request is formulated using HTTP (either *Post* or *Get*) in a form which mimics a remote method invocation (RMI); the server responds with a structured document of data in either XML (eXtensible Markup Language) or JSON (JavaScript Object Notation).

Flickr actually has quite a rich web service, with numerous functions covering a range of the site's capabilities. It also supports different web service data formats. In our project we'll use a lightweight ('RESTful') protocol to send the request, with the resulting data returned to us as an XML document. REST (REpresentational State Transfer) is becoming increasingly popular as a means of addressing web services; it generally involves less work than the heavily structured alternatives based around SOAP.

7.1.2 Getting registered with Flickr

Before we can go any further you'll need to register yourself as a Flickr developer, assuming you don't have an account already. This is so you can call their web service, which is a necessary part of this project. Signing up is relatively quick to do, and totally free for non-professional use. Once your account is created you'll be assigned a key (a long hexadecimal string) for use in any software you write accessing the service. The necessity for a key, it seems, is primarily to stop a single developer from flooding the site with requests.

```
http://www.flickr.com/services/api/
```

Go to the above web address and click the "Sign up" link at the head of the page, to begin the process of creating your account. The site will walk you through what you need to do, which shouldn't take too long. Once created, a "Your API keys" link will appear at the head of the page whenever you're logged in. Click it to view your developer account details, including the all-important key.

The site contains plenty of API documentation and tutorials. We'll only be using a tiny subset of the full API, but once you've seen an example of one web service call, it should be clear how to apply the documentation and add your own.

So, if you haven't already got a Flickr developer account, put the book down and register one right now, before you read any further. You can't run the project code without one, and the very next section will throw us straight into some web service coding.

7.2 Using a web service in JavaFX

At the heart of JavaFX's web service support are three classes. In the `javafx.io.http` package there's the `HttpRequest` class, used to actually make the HTTP request; and in `javafx.data.pull` there's `PullParser` and `Event`, used to parse the reply.

Our application also uses three classes itself: `FlickrService` will handle the request (using `HttpRequest`), `FlickrResult` will process the result (using `PullParser` and `Event`), and finally `FlickrPhoto` stores the details of the photos as they are pulled from the result.

In the sections ahead we'll examine each of these classes.

7.2.1 Calling the web service with HttpRequest

We'll start, naturally enough, with the `FlickrService`. You'll find it in listing 7.1 below. As before, the listing has been broken into stages to aid explanation.

Listing 7.1 FlickrService.fx (part 1)

```
package jfxia.chapter7;

import javafx.io.http.HttpRequest;
import javafx.data.pull.PullParser;
import java.io.InputStream;
import java.lang.Exception;

def REST:String = "http://api.flickr.com/services/rest/";                A

function createArgList(args:String[]) : String                           B
{   var ret="";                                                          B
    var sep="";                                                          B
    for(i in [0..<sizeof args step 2])                                   B
    {   ret="{ret}{sep}{args[i]}={args[i+1]}";                           B
        sep="&";                                                         B
    }                                                                    B
    return ret;                                                          B
}                                                                        B
// ** Part two below
```

A URL of web service

B Create HTTP query string from keys/values

We begin with one variable and one function, at script level. The variable, REST, is the base URL for the web service we'll be addressing. Onto this we'll add our request and its parameters. The function, `createArgList()`, is a useful utility for building the argument string appended onto the end of REST. If takes a sequence of key and value pairs and combines each into a query string using the format `key=value`, separated by ampersands.

Listing 7.2 shows the top of the `FlickrService` class itself.

Listing 7.2 FlickrService.fx (part 2)

```
// ** Part one above
public class FlickrService
{   public var apiKey:String;
    public var userId:String;
```

```
    public var photosPerPage:Integer = 10;
    public-read var page:Integer = 0;
    public var onSuccess:function(:FlickrResult);                          C
    public var onFailure:function(:String);                                C

    var valid:Boolean;                                                     D

    init                                                                   E
    {   valid = isInitialized(apiKey);                                     E
        if(not valid)                                                      E
            println("API key required.");                                  E
    }                                                                      E
// ** Part three below
```

C Callback functions
D Missing API key?
E Check for API key

At the head of the class we see various properties:

- The `apiKey` variable holds the developer key (the one associated with your Flickr account).
- The `userId` variable is for the account identifier of the person who's gallery we'll be viewing.
- The `photosPerPage` and `page` variables determine the page size (how many thumbs are fetched at once) and which page was previously fetched.
- Finally, `onSuccess` and `onFailure` are function types, permitting us to run code on the success or failure of our web service request.

In the `init` block we test for `apiKey` initialization; if it's unset we print an error message. A professional application would do something more useful with the error, of course, but for our project a simple error report like this will suffice (it keeps the class free of too much 'off-topic' detail).

We conclude the code with listing 7.3, next.

Listing 7.3 FlickrService.fx (part 3)

```
// ** Part two above
    public function loadPage(p:Integer) : Void
    {   if(not valid) throw new Exception("API key not set.");

        page = p;

        var args =                                                         F
        [   "method",    "flickr.people.getPublicPhotos",
            "api_key",    apiKey,
            "user_id",    userId,
            "per_page",   photosPerPage.toString(),
            "page",       page.toString()
        ];

        def http:HttpRequest = HttpRequest                                 G
        {   method: HttpRequest.GET;                                       H
            location: "{REST}?{createArgList(args)}";                      H
```

```
            onResponseCode: function(code:Integer)                          I
            {   if(code!=200 and onFailure!=null)
                    onFailure("HTTP code {code}");
            }
            onException: function(ex:Exception)                             J
            {   if(onFailure!=null)
                    onFailure(ex.toString());
            }
            onInput: function(ip:InputStream)                               K
            {   def fr = FlickrResult {};
                def parser = PullParser                                     L
                {   documentType: PullParser.XML;                           L
                    input: ip;                                              L
                    onEvent: fr.xmlEvent;                                   L
                };                                                          L
                parser.parse();                                             L
                parser.input.close();                                       L
                if(onSuccess!=null)  onSuccess(fr);
            }
        };
        http.enqueue();
    }
}
```

F Request arguments
G Web call
H Method and address
I Initial response
J I/O error
K Success!
L Create, and call, XML parser

Above we have the final part our service request code. The `loadPage()` function is where the action is at, it takes a page number and accesses the Flickr service to fetch the photo details for that page. After (double) checking the `apiKey`, and storing the selected page, it creates a sequence of key/value pairs to act as the arguments passed to the service call. The first list argument is the function we're actually calling on the web service, and the following arguments are parameters we're passing in.

Flickr's `flickr.people.getPublicPhotos` function returns a list of photos for a given user account, page by page. We need to pass in our own key, the ID of the person who's gallery we want to read, the number of photos we want back (the page size it should break the gallery up into), and which page we want. (See the web service API documentation for more details on this function.)

After the argument list we have the `HttpRequest` object itself. The HTTP request doesn't execute immediately; if it did it would hog the current thread and likely cause our UI to become unresponsive, as requests are often instigated during a UI event. Instead, when `enqueue()` is called, the network activity is pushed onto another thread, and we assign callbacks to run when there's something ready to act upon (see figure 7.2).

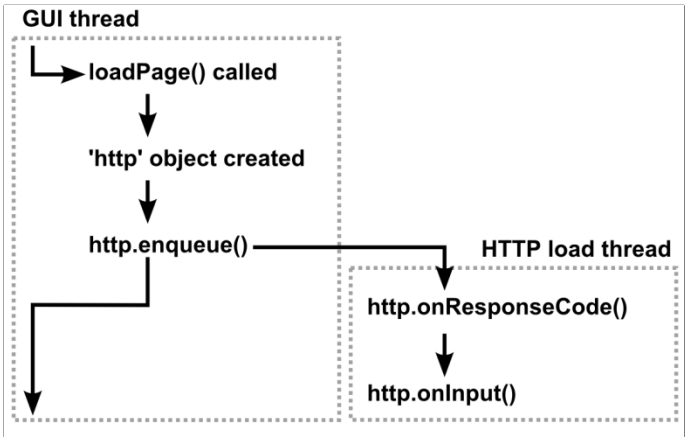


Figure 7.2 When enqueue() is called on a HttpRequest object, a second thread takes over and communicates its progress through callback events, allowing the GUI thread to get back to its work.

The `HttpRequest` request declaratively sets a number of properties. The `method` and `location` variables tell `HttpRequest` how and where to direct the HTTP call. To form the web address we use the script function `createArgList()`, turning the `args` sequence into a web-like query string, and append it to the REST base URL. The `onResponseCode`, `onException` and `onInput` event function types will be called at different stages of the request life cycle. The `HttpRequest` class actually has a host of different functions and variables to track the request state in fine detail (check the API docs), but typically we don't need such fine grained control.

The `onResponseCode` event is called when the initial HTTP response code is received ('200' means "OK", other codes signify different results), `onException` is called if there's an IO problem, while `onInput` is called when the result actually starts to arrive. The `onInput` call passes in a Java `InputStream` object, which we can assign a parser to.

And the JavaFX class `PullParser` is just such a parser. It reads either XML or JSON formatted data from the input stream, and breaks it down into a series of events. To receive the events we just need to register a function; but as our particular project needs to store some of the data being returned, I've written not just a single function, but an entire class (the `FlickrResult` class) to interact with it. And that's what we'll look at next.

7.2.2 Parsing XML with PullParser

Because we need somewhere to store the data we are pulling from the web service, we'll create an entire class to interact with the parser. That class is `FlickrResult`, taking each XML element as it is encountered, extracting data, and populating its variables. The class also houses a `FlickrPhoto` sequence, to store details for each individual photo.

Listing 7.4 is the first part of our class to process and store the information coming back from the web service.

Listing 7.4 FlickrResult.fx (part 1)

```
package jfxia.chapter7;

import javafx.data.pull.Event;
import javafx.data.pull.PullParser;
import javafx.data.xml.QName;


public class FlickrResult
{   public-read var stat:String;                                    A

    public-read var total:Integer;                                  B
    public-read var perPage:Integer;                                B
    public-read var page:Integer;                                   B
    public-read var pages:Integer;                                  B

    public-read var photos:FlickrPhoto[];                           C

    public def valid:Boolean =  bind (stat == "ok");                D
// ** Part two below
```

A Status message from service
B Gallery details
C Data for each photo in pages
D Request was successful?

Let's have a closer look at the details:

- The `stat` variable holds the success/failure of the response, as described in the reply. If Flickr can fulfill our request, we'll get back the simple message "ok".
- The `total` variable holds the number of photos in the entire gallery, `perPage` contains how many there are per page (should match the number requested), and `pages` details the number of available pages.
- Finally, `page` is the current page (again, it should match the one we requested).

The `valid` variable is a handy boolean for checking whether Flickr was able to respond to our request.

Following on we have listing 7.5, the second half of our parser class. It contains the actual code which responds to the `PullParser` events.

Listing 7.5 FlickrResult.fx (part 2)

```
// ** Part one above
    public function xmlEvent(ev:Event) : Void
    {   if(not (ev.type == PullParser.START_ELEMENT))               E
        {   return;
        }

        if(ev.level==0 and ev.qname.name == "rsp")                  F
        {   stat = readAttrS(ev,"stat");
        }
        else if(ev.level==1 and ev.qname.name == "photos")          G
        {   total = readAttrI(ev,"total");
            perPage = readAttrI(ev,"perpage");
```

```
            page = readAttrI(ev,"page");
            pages = readAttrI(ev,"pages");
        }
        else if(ev.level==2 and ev.qname.name == "photo")                H
        {   def photo = FlickrPhoto                                      I
            {   id: readAttrS(ev,"id");                                  I
                farm: readAttrS(ev,"farm");                              I
                owner: readAttrS(ev,"owner");                            I
                secret: readAttrS(ev,"secret");                          I
                server: readAttrS(ev,"server");                          I
                title: readAttrS(ev,"title");                            I
                isFamily: readAttrB(ev,"isfamily");                      I
                isFriend: readAttrB(ev,"isfriend");                      I
                isPublic: readAttrB(ev,"ispublic");                      I
            };                                                           I
            insert photo into photos;                                    I
        }
        else                                                             J
        {   println("{ev}");
        }
    }

    function readAttrS(ev:Event,attr:String) : String                    K
    {   def qn = QName{name:attr};
        return ev.getAttributeValue(qn) as String;
    }
    function readAttrI(ev:Event,attr:String) : Integer                   L
    {   return java.lang.Integer.parseInt(readAttrS(ev,attr));
    }
    function readAttrB(ev:Event,attr:String) : Boolean                   M
    {   return (readAttrI(ev,attr)!=0);
    }
}
```

E Not a start element? Exit!
F Top level, <rsp>
G 2nd level, <photos>
H 3rd level, <photo>
I Create and store photo object
J Didn't recognize element
K Read string attribute
L Read integer attribute
M Read boolean attribute

The function `xmlEvent()` is the callback invoked whenever a node in the XML document is encountered (note: *node* in this context does **not** refer to a scene graph node). Both XML and JSON documents are nested structures, forming a tree of nodes. JavaFX's parser walks this tree, firing an event for each node it encounters, with an `Event` object to describe the type of node (text or tag, for example), its name, its level in the tree, and so on.

So we're not working blind, below is an example of the sort of XML the web service might reply with. Each opening element tag, closing element tag, and loose text content inside an element, causes our event handler to be called.

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
```

```
  <photos page="1" pages="20" perpage="10" total="195">
    <photo id="3188821292" owner="12345678@N09" secret="cafebabe"
        server="3095" farm="4" title="Hello"
        ispublic="1" isfriend="0" isfamily="0" />
    <!-- Another nine photo elements appear here -->
  </photos>
</rsp>
```

Our XML handler is only interested in starting tags, that's why we exit if the node type isn't an element start.

The large if/else block parses specific elements. At level 0 we're interested in the `<rsp>` element, to get the status message (which we hope will be "ok"). At level 1 we're interested in the `<photos>` element, with attributes describing the gallery, page size, etc. Finally, at level 2, we're interested in the `<photo>` element, holding details of a specific photo on the page we're reading. Any other type of element, we simply print to the console (handy for debugging), then ignore.

The `<photo>` element is where we create each new `FlickrPhoto` object, with the help of three private functions for extracting named attributes from the tag in given data formats. Lets look at the `FlickrPhoto` class right now, in listing 7.6.

Listing 7.6 FlickrPhoto.fx

```
package jfxia.chapter7;

public def SQUARE:Number = 75;                                          A
public def THUMB:Number = 100;                                          A
public def SMALL:Number = 240;                                          A
public def MEDIUM:Number = 500;                                         A
public def LARGE:Number = 1024;                                         A

public class FlickrPhoto
{   public-init var id:String;                                          B
    public-init var farm:String;                                        B
    public-init var owner:String;                                       B
    public-init var secret:String;                                      B
    public-init var server:String;                                      B
    public-init var title:String;                                       B
    public-init var isFamily:Boolean;                                   B
    public-init var isFriend:Boolean;                                   B
    public-init var isPublic:Boolean;                                   B

    def urlBase:String = bind                                           C
        "http://farm{farm}.static.flickr.com/"
        "{server}/{id}_{secret}";
    public def urlSquare:String = bind "{urlBase}_s.jpg";               D
    public def urlThumb:String = bind "{urlBase}_t.jpg";                D
    public def urlSmall:String = bind "{urlBase}_m.jpg";                D
    public def urlMedium:String = bind "{urlBase}.jpg";                 D
    //public def urlLarge:String = bind "{urlBase}_b.jpg";              D
    //public def urlOriginal:String = bind "{urlBase}_o.jpg";           D
}
```

A Image pixel sizes

B Photo data, provided by the XML
C Base image address
D Actual image URLs

Each Flickr photo comes pre-scaled to various sizes. You'll note script level constants are used to describe the sizes of these images.

- A square thumbnail is 75x75 pixels.
- A regular thumbnail is 100 pixels on its longest side.
- A small image is 240 pixels on its longest side.
- A medium image is 500 pixels on its longest side.
- A large image is 1024 pixels on its longest side.
- The original image has no size restrictions.

Inside the class proper we find a host of `public-init` properties which store the details supplied via the XML response. The `farm`, `secret` and `server` variables are all used to construct the web address of a given image. The other variables should be self explanatory.

At the foot of the class we have the web addresses of each scaled image. The different sizes of image all share the same basic address, with a minor addition to the filename for each size (except for medium). We can use these addresses to load our thumbnails and full sized images. In our project we'll be working with the thumbnail and medium sized images only. Extra steps and permissions may be required to load the larger sized images using the web service API, so I've commented out the last two web addresses. The web service documentation explains how to get access to them.

And that's all we require to make, and consume, a web service request. All that's needed is some code to test it, but before we go there let's take a few moments to recap the process, to ensure we understand what's happening.

7.2.3 A quick recap

The process of calling a web service may seem a little convoluted. There's a lot of classes, function calls and event callbacks involved, so here's a quick blow-by-blow recap:

1. We formulate a web service request in `FlickrService`, using the service function we want to call plus its parameters.
2. Our `FlickrService` has two function types (event callbacks), `onSuccess` and `onFailure`, called upon the outcome of a web service request. We implement functions for these to deal with the data once it has loaded, or handle any errors.
3. Now everything is in place, we use JavaFX's `HttpRequest` to execute the request itself. It begins running *in the background*, allowing the current GUI thread to continue running unblocked.
4. If the `HttpRequest` fails, `onFailure` will be called. If we get as far as an `InputStream`, however, we create a parser (JavaFX's `PullParser`) to deal with the XML returned by the web service. The parser invokes an event callback

5. The callback function in `FlickrResult` parses each start tag in the XML. For each photo it creates and stores a new `FlickrPhoto` object.
6. Once the parsing is finished, execution returns to `onInput()` in `FlickrService`, which calls `onSuccess` with the resulting data.
7. In the onSuccess function we can now do whatever we want with the data loaded from the service.

Hopefully the above has made the process nice and clear. Now, at last, we need to actually see our web service in action.

7.2.4 Testing our web service code

Having spent the last few pages creating a set of classes to extract data from our chosen web service, we'll round off this part of the project with listing 7.7, showing how easy it is to use.

Listing 7.7 TestWS.fx

```
package jfxia.chapter7;

FlickrService
{   apiKey: "-";    // <== Your key goes here
    userId: "29803026@N08";
    photosPerPage: 10;

    onSuccess: function(res:FlickrResult)                          A
    {   for(photo in res.photos)                                   A
        {   FX.println("{photo.urlMedium}");                       A
        }                                                          A
    }                                                              A
    onFailure: function(s:String)                                  B
    {   println("{s}");                                           B
    }                                                              B
}.loadPage(1);

javafx.stage.Stage { visible: true; }                              C
```

A Success, print photo URLs
B Failure, print message
C Prevent termination

All we need is a small test program to create a web service request for photo details, and print out the URL of each medium sized image. To make the code work you'll need to supply the key you got when you signed up for a Flickr developer account.

As you can see, all that hard work paid off in the form of a nice and simple class we can use to get at our photo data.

Because the network activity takes place in the background, away from the main GUI thread, we need to stop the application from immediately terminating (before Flickr has time

to send any details back). We do this by creating a dummy window–its a crude solution, but effective. If all goes well the code should spit out, onto the console, a list of ten web addresses; one for each medium sized image in the first page of the gallery we accessed.

Now our network code is complete, we can get back to our usual habit of writing cool GUI code. In the next part of the project we're going to explore how to write styled widgets, one of JavaFX's hottest topics!

7.3 Creating a styled UI control in JavaFX

Java calls them *components*, I sometimes call them by the generic term *widgets*, but the new (official) JavaFX name for user interface elements is *controls*, it seems. Controls are buttons, text fields, sliders and other functional bits and pieces that enable us to collect input from the user, and display output in return. Unlike Java components, however, JavaFX controls are capable of using stylesheets. But what *are* stylesheets, and how can they work with JavaFX?

> **WARNING: UNCHARTED WATERS AHEAD**
>
> In the JavaFX 1.1 release, upon which this book was based, the controls library was unavailable. As I write, it is still in development, and the details in this part of the project are largely based on what little information there is out there. Because styled controls are such an important part of JavaFX, I didn't want to leave them out; even if their final form may not yet be quite set in concrete.
>
> Sources close to the JavaFX team have hinted that (broadly speaking) the techniques laid out in the following pages are likely to be close to the way the JavaFX controls API will work, once it is finalized. Some particular details may change, however. Readers are urged to search for fresh information which may have emerged since this chapter was written (January 2009), particularly any best practice guides to writing skins.

7.3.1 What is a stylesheet?

Back in the old days (when the web was in black and white) elements forming an HTML document determined both the logical meaning of their content, and how it would be displayed. For example, a <p> element indicated a given node in the DOM (Document Object Model) was of paragraph type. But marking the node as a paragraph also implied how it would be drawn on the browser page; how much whitespace would appear around it, how the text would flow, and so on. To counteract these presumptions new element types like <font> and were added to browsers to influence display. With no logical function in the document, these new elements polluted the DOM and made it impossible to render the document in different ways across different environments.

CSS (Cascading Style Sheets) was a means of fixing this problem, by separating the logical meaning of an element from the way it should be displayed. In a CSS file the web designer could specify the display settings of a paragraph, for example. The whitespace, color, text justification, and a host of other settings could be changed, all without need of

injecting extra style-specific elements into the HTML. These style sheet *rules* could be targeted at every node of a given type, at nodes having been assigned a given class, or a specific node carrying a given ID (see figure 7.3).

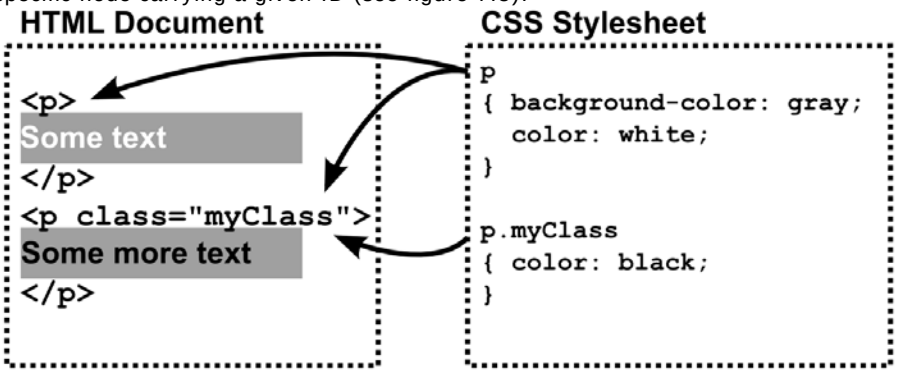
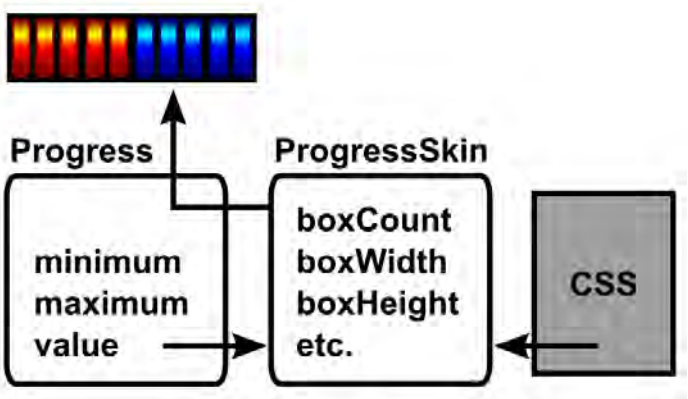


Figure 7.3 Two style rules and two HTML paragraph elements. The first rule applies to both paragraphs, while the second only applies to paragraphs of class “myClass”.

Untangling the logical structure of a document from how it should be displayed allowed the document to be shown in many different ways, simply by changing which stylesheet it was paired with. What worked for web documents could also work for graphical user interface controls, and this is what the JavaFX team hopes to achieve. If buttons, sliders, scrollbars, and other controls deferred to an external stylesheet for their display settings, artists and designers could change their look without having to write a single line of software code.



JavaFX actually allows both programmers and designers to get in on the style act. Each JavaFX control defers not to a stylesheet directly, but to a skin. Figure 7.4 shows this relationship diagrammatically. The skin is a JavaFX class which draws the control, and handles its mouse and keyboard events. The skin can also exposes various properties

(public instance variables) which JavaFX can then allow designers to change via a CSS-like file format.

Figure 7.4 The data from the control (Progress), and the CSS from the stylesheet, are combined inside the skin (ProgressSkin) to produce a displayable UI control. In this example, a progress bar.

In effect the control acts as the *model*, and the skin as the *view*/*controller* in the classic MVC paradigm used in many modern user interfaces. But the added JavaFX twist is that the skin can also be configured by a stylesheet.

Now we understand the theory, let's look at each part in turn as actual code.

7.3.2 Creating a control: the Progress class

We're going to write our own style compliant control from scratch, just to see how easy it is. The control we'll write will be a simple progress bar, like the one which might appear during a file download operation. The progress bar will take minimum and maximum bounds, plus a value, and display a series of boxes forming a horizontal bar, colored to show where the value falls in relation to its bounds.

Listing 7.8 shows the control class itself. This is the object other software will use when wishing to create a control declaratively. It subclasses the `Control` class from `javafx.scene.control`, a type of `CustomNode` designed to work with styling.

Listing 7.8 Progress,fx

```
package jfxia.chapter7;

import javafx.scene.control.Control;

public class Progress extends Control
{   public var minimum:Number = 0;                                    A
    public var maximum:Number = 100;                                  A
    public var value:Number = 50 on replace                           A
    {   if(value<minimum) { value=minimum; }                          A
        if(value>maximum) { value=maximum; }                          A
    };                                                                A

    override var skin = ProgressSkin{};                               B
}
```

A The control's data
B Override the skin

Our `Progress` class has three variables, the `maximum` and `minimum` determine the range of the progress (its high and low values), while `value` is the current setting within that range. We override the `skin` variable inherited from `Control` to assign a default skin object. The skin, recall, gives our control its face and processes user input. It's a key part of the styling process, so let's look at that class next.

7.3.3 Creating a skin: the ProgressSkin class

In itself the `Progress` class does nothing but hold the fundamental data of the progress meter control. Even though it's a `CustomNode` subclass, it defers all its display and input to the skin class. So, what does this skin class look like? It looks like listing 7.9, next.

Listing 7.9 ProgressSkin.fx (part 1)

```
package jfxia.chapter7;

import javafx.scene.Group;
import javafx.scene.control.Skin;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Paint;
import javafx.scene.paint.Stop;
import javafx.scene.shape.Rectangle;


public class ProgressSkin extends Skin
{   public var boxCount:Integer = 10;                                   A
    public var boxWidth:Number = 7;                                     A
    public var boxHeight:Number = 20;                                   A
    public var boxStrokeWidth:Number = 1;                               A
    public var boxCornerArc:Number = 3;                                 A
    public var boxHGap:Number = 1;                                      A
                                                                        A
    public var boxUnsetStroke:Paint = Color.DARKBLUE;                   A
    public var boxUnsetFill:Paint = makeLG(Color.CYAN,                  A
        Color.BLUE,Color.DARKBLUE);                                     A
    public var boxSetStroke:Paint = Color.DARKGREEN;                    A
    public var boxSetFill:Paint = makeLG(Color.YELLOW,                  A
        Color.LIMEGREEN,Color.DARKGREEN);                               A

    def boxValue:Integer = bind                                         B
    {   var p:Progress = control as Progress;                           B
        var v:Number = (p.value-p.minimum) /                            B
            (p.maximum-p.minimum);                                      B
        (boxCount*v) as Integer;                                        B
    }                                                                   B
// ** Part two below
```

A These properties can be styled with CSS

B How many boxes to highlight?

Listing 7.9 is the opening of our skin class, `ProcessSkin`. The `Progress` class, recall, is effectively the model, and this class is the view/controller in the classic MVC scheme. It subclasses `javafx.scene.control.Skin`, allowing it to be used as a skin.

The properties at the top of the class are all exposed so that they can be altered by a stylesheet. They perform various stylistic functions.

- The `boxCount` variable determines how many progress bar boxes should appear on screen.

- The `boxWidth` and `boxHeight` variables hold the dimensions of each box, while `boxHGap` is the pixel gap between boxes.
- The variable, boxStrokeWidth, is the size of the trim around each box, and boxCornerArc is the radius of the rounded corners.
- Finally, for the public interface, we have two pairs of variables which colorize the control. The first pair is `boxUnsetStroke` and `boxUnsetFill`, the trim and body colors for *switched off* boxes; the second pair is (unsurprisingly) `boxSetStroke` and `boxSetFill`, and they do the same thing for *switched on* boxes. The `makeLG()` function is a convenience for creating gradient fills, we'll see it in the concluding part of the code.
- The private variable `boxValue` uses the data in the `Progress` control to work out how many boxes should be switched on. The reference to `control` is a variable inherited from its parent class, `Skin`, allowing us to read the current state of the control (the model) the skin is plugged into.

One thing of particular note in the above: the stroke and fill properties are `Paint` objects, not `Color` objects. Why? Quite simply, the former allows us to plug in a gradient fill or some other complex pattern in, while the latter would only support a flat color. And, believe it or not, JavaFX's styles support actually extends all the way to patterned fills.

Moving on, the concluding part of the source code (listing 7.10) shows how these variables are used to construct the progress meter.

Listing 7.10 ProgressSkin.fx (part 2)

```
init
{   scene = HBox                                                    C
    {   spacing: bind boxHGap;
        content: bind for(i in [0..<boxCount])
        {   Rectangle
            {   width: bind boxWidth;                               D
                height: bind boxHeight;                             D
                arcWidth: bind boxCornerArc;                        D
                arcHeight: bind boxCornerArc;                       D
                strokeWidth: bind boxStrokeWidth;                   D
                stroke: bind                                        E
                    if(i<boxValue) boxSetStroke                     E
                    else boxUnsetStroke;                            E
                fill: bind                                          F
                    if(i<boxValue) boxSetFill                       F
                    else boxUnsetFill;                              F
            };
        }
    };
}

function makeLG(c1:Color,c2:Color,c3:Color) : LinearGradient        G
{   LinearGradient
    {   endX: 0;  endY: 1;  proportional: true;
        stops:
        [   Stop { offset:0;    color: c3; } ,
```

```
                Stop { offset:0.25; color: c1; } ,
                Stop { offset:0.50; color: c2; } ,
                Stop { offset:0.85; color: c3; }
            ];
        };
    }
}
```

C Assign to scene in Skin class
D Bind visible properties
E Bind trim color to boxValue
F Bind body color to boxValue
G Gradient paint from three colors

We see how the style variables are used to form a horizontal row of boxes. An inherited variable from `Skin`, named `scene`, is used to record the skin's scene graph. Any node plugged into `scene` becomes the corporeal form (physical body) of the control the skin is applied to. In our case, we've a heavily bound sequence of `Rectangle` objects, each tied to the instance variables of the class. This sequence is all it takes to create our progress bar.

At the end of the listing is the `makeLG()` function, a convenience for creating the `LinearGradient` paints used as default values for the box fills.

All that remains, now we've seen the control and its skin, is to take a look at the style document itself.

7.3.4 Using our styled control with a CSS document

The Progress and ProgressSkin classes create a new type of control, capable of being configured through an external stylesheet document. Now its time to see how our new control can be used, and manipulated.

Listing 7.11 is a test program for trying out our new control. It creates three examples: the first is a regular `Progress` control without any *class* or *ID* (note, in this context the word *class* refers to the CSS style class, and has nothing to do with the JavaFX Script class), the second is another `Progress` control with an ID ("`testID`"), and the final `Progress` has been assigned to a style class ("`testClass`").

Listing 7.11 TestCSS.fx

```
package jfxia.chapter7;

import javafx.animation.KeyFrame;
import javafx.animation.Interpolator;
import javafx.animation.Timeline;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;

var val:Number = 0;
Stage
{   scene: Scene
    {   var vb:VBox;
        content: vb = VBox
        {   spacing:10;
```

```
            translateX: 5;  translateY: 5;
            content:
            [   Progress                                                    A
                {   minimum: 0;  maximum: 100;
                    value: bind val;
                } ,
                Progress                                                    B
                {   id: "testId";
                    minimum: 0;  maximum: 100;
                    value: bind val;
                } ,
                Progress                                                    C
                {   styleClass: "testClass";
                    style: "boxSetStroke: white";
                    minimum: 0;  maximum: 100;
                    value: bind val;
                }
            ];
        };
        stylesheets: [ "{__DIR__}Test.css" ]                                D
        fill:  Color.BLACK;
        width: 225;
        height: 110;
    };
    title: "CSS Test";
    visible: true;
};

Timeline
{   repeatCount: Timeline.INDEFINITE;
    autoReverse: true;
    keyFrames:                                                              E
    [   at(0s)   { val => 0 } ,                                             E
        at(0.1s) { val => 0 tween Interpolator.LINEAR } ,                   E
        at(0.9s) { val => 100 tween Interpolator.LINEAR } ,                 E
        at(1s)   { val => 100 }                                             E
    ];                                                                      E
}.play();
```

A Plain control, no id or class
B Control with id
C Control with class
D Assign stylesheets
E Run val backwards and forwards

Note how the final progress bar also assigns its own local style for the `boxSetStroke`? This is important, as we'll see in a short while.

Figure 7.5 shows the progress control on screen. All three `Progress` bars are bound to the variable `val`, which the `Timeline` at the foot of the code repeatedly increases and decreased (with a slight pause at either end), to make the bars shoot up and down from minimum to maximum.

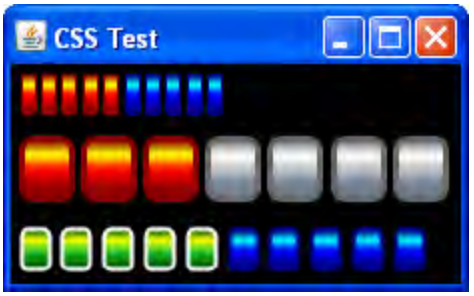


Figure 7.5 Three examples of our progress bar in action.

The key part of the code lies in the `stylesheets` property of `Scene`. This is where we plug in our list of CSS documents (just the one, in our example) using their URLs. This particular example expects our docs to sit next to the TestCSS bytecode files. The `__DIR__` built-in variable returns the URL of the current class file, recall.

If you downloaded the project's source code you'll find the CSS file nested inside the `res` directory, off from the project's root. When you build the project, make sure this file gets copied into the build directory, next to the `TessCSS` class file.

Classes and IDs

In style sheets, classes and IDs perform similar functions, but for use in different ways. Assigning an ID to a node in a document means it can be targeted specifically. IDs are supposed to be unique names, only appearing once in a given document. They can be used for more than just targeting stylesheet rules..

What happens if we need to apply a style, not to a specific DOM node, but to an entire subset of nodes? To do this we use a class, a non-unique identifier, designed primarily to as a *type* identifier onto which CSS rules can bind.

Finally, it is time for the moment you've all been waiting for: the grand unveiling of the CSS file which styles our component.

Listing 7.12 shows the three style rules we've created for our test program. Remember, this file must be copied into the build directory inside the `jfxia.chapter7` package, so it's next to the `TestCSS` class.

Listing 7.12 Test.css

```
"jfxia.chapter7.Progress"
{   boxSetStroke: darkred;
    boxSetFill: linear (0%,0%) to (0%,100%) stops
        (0.0,darkred), (0.25,yellow), (0.50,red), (0.85,darkred);
    boxUnsetStroke: darkblue;
    boxUnsetFill: linear (0%,0%) to (0%,100%) stops
        (0.0,darkblue), (0.25,cyan), (0.50,blue), (0.85,darkblue);
```

```
}

"jfxia.chapter7.Progress"#testId
{   boxWidth: 25;  boxHeight: 30;
    boxCornerArc: 12;  boxStrokeWidth: 3;
    boxCount: 7;
    boxHGap: 1;

    boxUnsetStroke: dimgray;
    boxUnsetFill: linear (0%,0%) to (0%,100%) stops
        (0%,dimgray), (25%,white), (50%,silver), (75%,slategray);
}

"Progress".testClass
{   boxWidth: 14;
    boxCornerArc: 7;
    boxStrokeWidth: 2;
    boxHGap: 3;

    boxSetStroke: darkgreen;
    boxSetFill: linear (0%,0%) to (0%,100%) stops
        (0.0,darkgreen), (0.25,yellow), (0.50,limegreen), (0.85,darkgreen);
}
```

The first rule targets all instances of `jfxia.chapter7.Progress` controls. The settings in its body are applied to the skin of the control. The second is far more specific: like the first, it applies to `jfxia.chapter7.Progress` controls, but only to that particular control with the ID of "`testID`". The final rule targets the `Progress` class again (this time omitting the package prefix, just to show it's not necessary if the class name alone is not ambiguous), but it applies itself to any such control belonging to the style class "`testClass`".

If multiple rules match a single control, the styles from all rules are applied in a strict order. It starts with the generic (no class, no id) rule, continuing with the class rule, then the specific ID rule, and finally any style plugged directly into the object literal inside the JavaFX Script code. Remember that explicit style assignment I said was important a couple of pages back? That was an example of overruling the CSS file with a style written directly into the JFX code itself. The styles for each matching rule are applied in order, with later styles overwriting the assignments of previous styles.

And in case you're wondering, if absolutely nothing matches the control, the default styles defined in the skin class it self, ProgressSkin in our case, remain untouched. It's important, therefore, to ensure your skins always have sensible defaults.

You'll note how the class name is wrapped in quotes? If you were wondering, this is simply to stop the dots in the name from being misinterpreted as CSS style class separators, like the one immediately before the name "`testClass`".

Inside the body of each style rule we see the skin's public properties being assigned. The majority of these assignments are self explanatory. Variables like `boxCount`, `boxWidth` and `boxHeight` all take integer numbers, and color variables can take CSS color definitions or names, but what about the strange `linear` syntax?

Cascading Style Sheets

In this book we don't have the space to go into detail about the format of CSS stylesheet, on which JavaFX stylesheets are firmly based. CSS is a World Wide Web Consortium specification, and the W3C web site has plenty of documentation on the format:

`http://www.w3.org/Style/CSS/`

7.3.5 Further CSS details

The exact nature of how CSS will interact with JavaFX skins is still not finalized as this chapter is being written, yet already several JFX devotees have dug deep into the class files and discovered some of the foundations laid for when the controls API does arrive.

As this information may be subject to change before the API's final release, we'll look at just a couple of examples to get an idea of what's available. A quick internet search will no doubt reveal further styling options, although hopefully we won't have to wait too long for the final API, with it's official documentation.

Reproduced below is one of the linear paint examples from the featured stylesheet.

```
boxUnsetFill: linear (0%,0%) to (0%,100%) stops
    (0.0,dimgray), (0.25,white), (0.50,silver), (0.75,slategray);
```

The example creates, as you might expect, a `LinearGradient` paint starting in the top left corner (0% of the way across the area, and 0% of the way down), and ending in the bottom left (0% of the way across, 100% of the way down). This results in a straight forward vertical gradient. To define the color stops on we use a comma separated list of position/color pairs in parenthesis. For the positions we could use percentages again, but for the sake of variety the example shows an alternative fraction based scale, from 0 to 1. The colors are regular CSS color definitions (see the W3C's documentation for details).

The stylesheet in our example is tied directly to the `ProgressSkin` we created for our `Progress` control. The setting it changes are the publicly accessible variables inside the skin class. But was can go further than just tweaking the skin's variables, we can actually replace the entire skin class.

```
“Progress”#testID
{   skin: jfxia.chapter7.AlternativeProgressSkin;
}
```

The fragment of stylesheet above sets the skin itself, providing an alternative class to the `ProgressSkin` we installed by default. Obviously we haven't actually written such a class, this is just a demonstration. The style rule targets a specific example of the `Progress` control, with the ID “testID”; although if we removed the ID specifier from the rule it would target all `Progress` controls.

The `AlternativeProgressSkin` class would have its own public interface, with its own variables which could be styled. For this reason, the rule should be placed above all other

rules which style the variables of the `AlternativeProgressSkin` (indeed some commentators have noted it works best when placed in a separate rule, all on its own).

So much for stylesheets! We've written a web service, and now we have a styled control. It's time to pull them together into an actual photo viewer application, but there's still some surprises to come. So far in the book animations have been restricted to creating our own time lines, but JavaFX comes with a powerful library of ready-to-go animation transitions, and we'll see plenty of diverse examples in the final part of the project, up next.

7.4 Pulling it all together: the PhotoViewer application

In this, the final part of the project, we're going to use the web service classes we developed earlier, and the styled progress bar, in an application to throw photos on screen. The application will be full screen–that is to say it will not run in a window on the desktop, but will take over the whole display. It will also use transitions to perform its movement and other animated effects.

The application has a bar or thumbnails along its foot, combined with three buttons, demonstrated in figure 7.6. One button moves the thumbnails forward by a page, another moves it back, and the final button clears the main display area (or *desktop*, as I'm calling it) of photos. As we move over the thumbnails in the bar, the associated title text (which the web service gave us) is displayed.

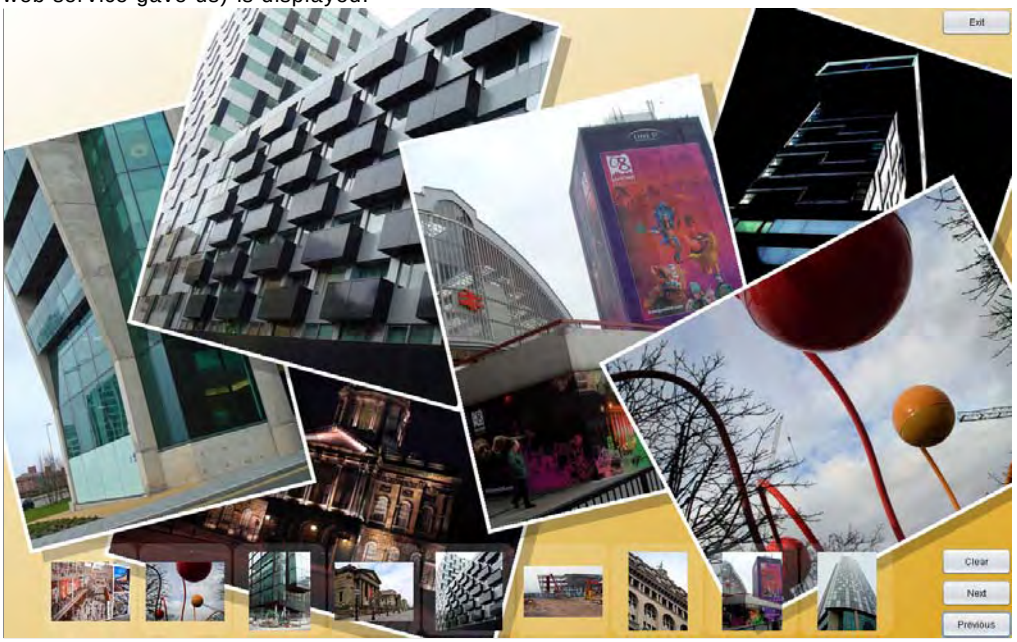


Figure 7.6 Photos selected from the thumbnail bar fly into the 'desktop'.

To get a photo onto the desktop we merely click its thumbnail, to see it spin dramatically onto the display scaled to full size. Initially we scale the tiny thumbnail up to the size of the photo, while we wait for the higher resolution image to be loaded. As we wait, the styled progress bar appears in the corner of the photo, showing how much of the image has arrived over the network. When the high resolution image finally finishes loading it replaces the scaled up thumbnail, and the progress bar vanishes.

We can click and drag individual images to move them around the desktop, or we can click on an empty part of the desktop and drag, to move all the images at once.

The application itself is constructed from further two classes, our largest yet at over two hundred lines apiece. But don't worry, they still contain plenty of fresh JavaFX goodness for us to explore. As usual, they've been broken up into parts to aid explanation. We begin with we'll look at the class which handles the thumbnail bar.

7.4.1 Displaying thumbnails from the web service: the GalleryView class

The `GalleryView` class is the visual component which deals with the Flickr web service, and presents a horizontal list of thumbnail based on the data it extracts from the service. Figure 7.7 shows the specific part of the application we're building.

Figure 7.7 The custom scene graph node we are creating.

So that's what we want it to look like, let's dive straight into the source code with listing 7.13, below. Here we see the variables in the top part of our `GalleryView` class.

Listing 7.13 GalleryView.fx (part 1)

```
package jfxia.chapter7;

import javafx.animation.Interpolator;
import javafx.animation.transition.TranslateTransition;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.scene.text.Text;

package class GalleryView extends CustomNode
{   def thumbWidth:Number = FlickrPhoto.THUMB;                          A
    def thumbHeight:Number = FlickrPhoto.THUMB;                         A
    def thumbBorder:Number =  10;                                       A
```

```
    public-init var apiKey:String;                                    B
    public-init var userId:String;                                    B
    package var width:Number = 0;
    package def height:Number = thumbHeight + thumbBorder*2;
    package var action:function(:FlickrPhoto,                         C
        :Image,:Number,:Number);

    public-read var page:Integer = 1                                  D
        on replace                                                    D
        {   loadPage();                                               D
        };
    public-read var pageSize:Integer =
        bind
        {   def aw:Number = width;                                    E
            def pw:Number = thumbWidth + thumbBorder*2;               E
            (aw/pw).intValue();                                       E
        }
        on replace
        {   if(pageSize>0)  createUI();                               F
            loadPage();

        };
    var service:FlickrService;                                        G
    var result:FlickrResult;                                          G
    var thumbImages:Image[];                                          G

    var topGroup:Group = Group{};                                     H
    var thumbGroup:Group;                                             H
    var textDisplay:Text;                                             H
// ** Part two below
```

A Handy constants
B Flickr details
C Thumbnail clicked event
D New page means thumbs reload
E Width determines thumbnail count
F Rebuild scene graph on change
G Flickr classes and fetched thumbs
H handy scene graph stuff

At its head we see a few handy constants being defined: the maximum dimensions of a thumbnail (simply brought in from the `FlickrPhoto` class for convenience), and the gap between each thumbnail. The other variables are as follows:

- The `apiKey` and `userId` variables should be familiar from the first part of the project. We set these when we create a `GalleryView`, so it can access the Flickr service.
- The `height` of the bar we can calculate from the size of the thumbnails and their surrounding frames, but the `width` is set externally (by using the size of the screen).
- The `action` function type holds the callback we use to tell the outside application that a thumbnail has been clicked. The parameters are the `FlickrPhoto`

associated with this thumbnail, the thumb `Image` already downloaded, and the x/y location of the thumbnail in the bar.

- The `page` and `pageSize` variables control which page of thumbnails is loaded, and how many thumbnails are on that page. Changing either causes an access to the web service to fetch fresh data, which is why both have an `on replace` block. A change to `pageSize` will also cause the contents of the scene graph to be rebuilt, using the function we created for this very purpose. The page size is determined by the number of thumbnails we can fit inside the `width`, which explains the bind.
- The private variables `service`, `results` and `thumbImages` all relate directly to the web service. The first is the interface we use to load each page of thumbnails, the second is the result of the last page load, and finally we have the actual thumbnail images themselves.
- Finally, private variables `topGroup`, `thumbGroup` and `textDisplay` are all parts of the scene graph which need manipulating in response to events.

Now we'll turn to the actual code which initializes those variables. Listing 7.14 sets up the web service and returns the top level node of our bit of the scene graph.

Listing 7.14 GalleryView.fx (part 2)

```
// ** Part one above
    init
    {   service = FlickrService                                    I
        {   apiKey: bind apiKey;
            userId: bind userId;
            photosPerPage: bind pageSize;
            onSuccess: function(res:FlickrResult)                  J
            {   result = res;                                      J
                assignThumbs(result);                              J
            }                                                      J
            onFail: function(s:String)                             K
            {   println("{s}");                                    K
            }                                                      K
        };
    }

    override function create() : Node                              L
    {    topGroup;                                                 L
    }                                                              L
// ** Part three below
```

I Flickr web service class
J Do this on success
K Do this on failure
L Return scene graph node

The code shouldn't need too much explanation. We register two functions with the `FlickrService` class, `onSuccess` will run when data has been successfully fetched, and `onFailure` will run if it hits a snag. In the case of a successful load, we store the `result` object so we can use its data later, and call a private function (see later for the code) to copy

the URLs of the new thumbnails out of the `result` and into the `thumbImage` sequence, causing them to start loading.

Listing 7.15, the third part of the code, will take us to the scene graph for this class.

Listing 7.15 GalleryView.fx (part 3)

```
// ** Part two above
    function createUI() : Void                                          M
    {   def sz:Number = thumbWidth + thumbBorder*2;
        var thumbImageViews:ImageView[] = [];

        textDisplay = Text                                              N
        {   font: Font { size: 30; }
            fill: Color.BROWN;
            translateX: bind
                (width-textDisplay.layoutBounds.width)/2;
        }

        thumbGroup = Group                                              O
        {   content: for(i in [0..<pageSize])
            {   var iv:ImageView = ImageView
                {   translateX: bind i*sz + thumbBorder +               P
                        (if(iv.image==null) 0                           P
                        else (thumbWidth-iv.image.width)/2);            P
                    translateY: bind thumbBorder +                      P
                        (if(iv.image==null) 0                           P
                        else (thumbHeight-iv.image.height)/2);          P
                    fitWidth: thumbWidth;                               Q
                    fitHeight: thumbHeight;                             Q
                    preserveRatio: true;                                Q

                    image: bind if(thumbImages!=null)                   R
                        thumbImages[i] else null;                       R
                };
                insert iv into thumbImageViews;
                iv;
            };
        };

        def frameGroup:Group = Group                                    S
        {   content: for(i in [0..<pageSize])
            {   def r:Rectangle = Rectangle
                {   translateX: i*sz + 2;
                    width: sz-4;  height: sz - 4;
                    arcWidth: 25;  arcHeight: 25;
                    opacity: 0.15;
                    fill: Color.WHITE;

                    onMouseEntered: function(ev:MouseEvent)             T
                    {   r.opacity = 0.35;
                        if(result!=null and
                            i<(sizeof result.photos))
                        {    textDisplay.content =
                                result.photos[i].title;
                        }
                    }
```

```
                    onMouseExited: function(ev:MouseEvent)                U
                    {   r.opacity = 0.15;
                        textDisplay.content = "";
                    }
                    onMouseClicked: function(ev:MouseEvent)               V
                    {   if(action!=null)
                        {   def f = result.photos[i];
                            def t = thumbImages[i];
                            def v = thumbImageViews[i];
                            def x:Number = v.translateX;
                            def y:Number = v.translateY;
                            action(f, t,x,y);
                        }
                    }
                };
            };
        };

        topGroup.translateX = (width - pageSize*sz) / 2;                  W
        topGroup.content =                                                X
        [    frameGroup , thumbGroup , textDisplay                        X
        ];                                                                X
    }
// ** Part four below
```

M Create actual scene graph
N Photo title banner text
O Sequence of thumbnail images
P Center align thumb
Q Resize image
R Use image, if available
S Background rectangle
T Change opacity, show title text
U Change opacity, clear title text
V Fire action event
W Center gallery view
X Create top level node

Listing 7.15 is the real meat of the scene graph. Whenever the `pageSize` variable is changed (assuming it's not the initial zero assignment) the `createUI()` function runs to re-populate the `topGroup` node, which is the root of our scene graph fragment.

The code constructs three layers of node, as shown in figure 7.8. We separate the images from their background frames, so that we can animate them separately. Whenever a new page of thumbnails is loaded, the current thumbnails will fall out of view–we can only do this if they are grouped independently of their framing rectangles.

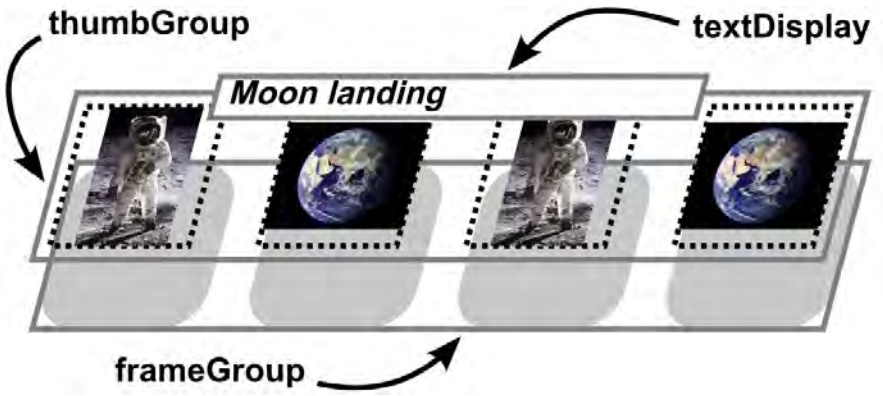


Figure 7.8 The thumbnail bar scene graph is made up on three core components, a group of frame rectangles in the background, a group of images over it, and finally a text node to display the thumb titles.

The `textDisplay` is used to show the title of each image as the mouse passes over it. We build the `thumbGroup` by adding an image at a time, scaled to fit the space available (which shouldn't be necessary if Flickr deliver the thumbs in the size we expect, but just in case this changes...). We store each `Image` as we create it, because we'll need access to this thumbnail in the `onMouseClicked()`event handler.

The frameGroup is where we put all the mouse handling code. A rollover causes the photo's text to be displayed (entered) or cleared (exited), and the frame's opacity is changed. When the mouse is clicked it triggers an `action` event, with the appropriate `FlickrPhoto` object, its thumbnail image (we stored it earlier), and the co-ordinates within the bar, all bundled up and passed to the function assigned to `action`.

Almost there! Only one block of code left, listing 7.16, and it is has a surprise up its sleeve.

Listing 7.16 GalleryView.fx (part 4)

```
// ** Part three above
    package function next() : Void { setPage(page+1); }
    package function previous() : Void { setPage(page-1); }
    function setPage(p:Integer) : Void                                    Y
    {   if(result!=null)
        {   page =
                if(p<=0)  result.pages
                else if(p>result.pages)  1
                else  p;
        }
    }

    function loadPage() : Void                                            Z
```

```
    {   if(service!=null and pageSize>0 and page>0)
        {   TranslateTransition                                          AA
            {   node: thumbGroup;                                        BB
                byX: 0;  byY: height;                                    CC
                interpolate: Interpolator.EASEIN;
                duration: 0.5s;
                action: function()                                       DD
                {   unassignThumbs();
                    thumbGroup.translateY = 0;
                    service.loadPage(page);
                }
            }.play();                                                    EE
        }
    }
    function assignThumbs(res:FlickrResult) : Void                       FF
    {   thumbImages = for(i in [0..<sizeof res.photos])
        {   var im = Image
            {   url: res.photos[i].urlThumb;
                backgroundLoading: true;
            };
        };
    }
    function unassignThumbs() : Void                                     GG
    {   thumbImages =  [];
    }
}
```

Y Make sure page is within range
Z Load page of thumbs
AA Move...
BB ...this node...
CC ...by this amount.
DD Do this, when finished
EE Run transition
FF URLs from result into ImageViews
GG Clear thumb image

The final part of our mammoth `GalleryView` class begins with three functions for manipulating the page variable, ensuring it never strays outside the acceptable range (Flickr starts its page numbering at 1, in case you were wondering). The `next()` and `previous()` functions will be called by outside classes to cause the gallery to advance or retreat.

Skipping over `loadPage()` for now (we'll come back to it in a moment), the `assignThumbs()` and `unassignThumbs()` functions do what their name suggests. The first takes a `FlickrResult`, as retrieved from the web service, and populates the `ImageView` nodes in the thumbnail bar with fresh `Image` content. The second clears those same images, to remove the thumbnails from the bar.

And now for the `loadPage()` function: the code ultimately responsible for responding to each request to fetch a fresh page of thumbnails from the web service. The entire function is based around a strange new type of operation called a *transition*. We've yet to see a transitions in any of the projects so far, so let's stop and examine it in detail.

7.4.2 The easy way to animate: transitions

So far in the book, whenever we wanted to animate something we had to build a `Timeline` and use it to manipulate the variables was wanted to change. It doesn't take a rocket scientist to realize there's a handful of common node attributes (location, rotation, opacity, etc.) which are frequent targets for animation. We could save a lot of unnecessary boilerplate if we created a library of pre-build animation classes, pointed them at the nodes we wanted to animate, then just let them get on with the job.

If you haven't guessed by now, this is what *transitions* are.

Let's have another look at the code in the last section:

```
TranslateTransition
{   node: thumbGroup;
    byX: 0;  byY: height;
    interpolate: Interpolator.EASEIN;
    duration: 0.5s;
    action: function()
    {   unassignThumbs();
        thumbGroup.translateY = 0;
        service.loadPage(page);
    }
}.play();
```

The `TranslateTransition` is all about movement. It has a host of different configuration options which allow us to move a node *from* a given point, *to* a given point, or *by* a given distance. In our example we're moving the entire thumbnail group downward by the height of the thumbnail bar; which (one hopes) would have the effect of sending the thumbnails off the bottom of the screen.

When the transition is over, we have the opportunity to run some code. That's the purpose of the `action` function type. When this code runs we know the thumbnails will be off screen, so we un-assign them to get rid of the current images. Then we move the group we just animated back to its starting position. And finally, we ask the web service to load a fresh page of thumbnails. This call will return very quickly as the loading is done on another thread, you'll recall (right?) Back in listing 7.14 we saw that the web service invokes `assignThumbs()` when its data has successfully loaded, so the images we just deleted with `unassignThumbs()` should start to be repopulated once the web service code is finished.

And finally, in case you haven't realized it, `play()` fires the transition into action.

7.4.3 The main photo desktop: the PhotoViewer class

To round off our project we're going to look at clicking and dragging bits of the scene graph, we're going to actually use that swanky progress bar we created a while back, and we'll be playing with even more types of transition. And it all takes place in the `PhotoViewer` class, of which Listing 7.17 is the first part.

Listing 7.17 PhotoViewer.fx (part 1)

```
package jfxia.chapter7;
```

```
import javafx.animation.Interpolator;
import javafx.animation.transition.FadeTransition;
import javafx.animation.transition.ParallelTransition;
import javafx.animation.transition.ScaleTransition;
import javafx.animation.transition.RotateTransition;
import javafx.animation.transition.TranslateTransition;
import javafx.ext.swing.SwingButton;
import javafx.geometry.Rectangle2D;
import javafx.scene.*;
import javafx.scene.image.*;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.*;
import javafx.scene.layout.VBox;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

import java.lang.Math;
import java.lang.System;
import java.util.Random;

def buttonW:Number = 90;                                                    A
def buttonH:Number = 35;                                                    A
def maxImages:Integer = 6;                                                  A
def rand:Random = new Random(System.currentTimeMillis());                   A

var sc:Scene;

def exitButton = SwingButton                                                B
{   text: "Exit";
    translateX: bind sc.width - buttonW;
    width: buttonW;  height: buttonH;
    action: function() { FX.exit(); }
};

def mouseHandler:Rectangle = Rectangle                                      C
{   var dragX:Number;
    var dragY:Number;
    width: bind sc.width;
    height: bind sc.height;
    opacity: 0;
    onMousePressed: function(ev:MouseEvent)                                 D
    {   dragX=ev.x;  dragY=ev.y;                                            D
    }                                                                       D
    onMouseDragged: function(ev:MouseEvent)                                 E
    {   var idx:Integer = 0;                                                E
        for(i in desktopGroup.content)                                      E
        {   def v:Number = 1.0 + idx/3.0;                                   E
            i.translateX += (ev.x-dragX) * v;                               E
            i.translateY += (ev.y-dragY) * v;                               E
            idx++;                                                          E
        }                                                                   E
        dragX=ev.x;  dragY=ev.y;                                            E
    }                                                                       E
}
def desktopGroup:Group = Group {}                                           F
```

```
// ** Part two below
```
A Constants and utility objects
B Swing exit button
C Capture desktop click events
D Remember initial position
E Move all photos with parallax
F Photos added here

At the head of the source we define a few constants, such as the button size and maximum number of photos on the desktop. We also create an instance of Java's random number class, which we'll use to add a degree of variety to the animations.

To get the dimensions of the screen we need a reference to its `Scene`, which is what `sc` is for. The exit button, imaginatively named `exitButton`, comes next. Then we define an invisible `Rectangle`, for capturing desktop mouse clicks and drag events. When the user clicks on an empty part of the desktop, the event will be directed to this shape. On button down, it stores the x/y position. As drag events come in it moves all the nodes inside a `Group` called `desktopGroup`. This group is where all the desktop photos are stored. As they are moved the code adds a small scaling factor, so later (higher) photos move more than earlier (lower) ones, creating a cool parallax effect. (Actually, my friend Adam wasn't so keen on the parallax effect when I showed it to him, but what does he know?!?)

Moving on, and listing 7.18 shows us how our `GalleryView` class in action.

Listing 7.18 PhotoViewer.fx (part 2)

```
// ** Part one above
def galView:GalleryView = GalleryView                                    G
{   translateY: bind (sc.height-galView.height);                         H
    width: bind sc.width-100;                                            H

    apiKey: "-";  // <== Your key goes here                              I
    userId: "29803026@N08";                                              I

    action: function(ph:FlickrPhoto ,im:Image,x:Number,y:Number)         J
    {   def pv:Node = createPhoto(ph,im);
        def pvSz:Rectangle2D = pv.layoutBounds;
        def endX:Number = (sc.width-pvSz.width) / 2;
        def endY:Number = (sc.height-pvSz.height) / 2;
        ParallelTransition                                               K
        {   node: pv;
            content:
            [   TranslateTransition                                      L
                {   fromX: galView.translateX + x;                       L
                    fromY: galView.translateY + y;                       L
                    toX: endX;                                           L
                    toY: endY;                                           L
                    interpolate: Interpolator.EASEIN;                    L
                    duration: 0.5s;                                      L
                } ,                                                      L
                ScaleTransition                                          M
                {   fromX: 0.25;  fromY: 0.25;                           M
                    toX: 1.0;  toY: 1.0;                                 M
                    duration: 0.5s;                                      M
                },                                                       M
```

```
                RotateTransition                                        N
                {   fromAngle: 0;                                       N
                    byAngle: 360 + rand.nextInt(60)-30;                 N
                    duration: 0.5s;                                     N
                }                                                       N
            ];
        }.play();

        insert pv into desktopGroup.content;                            O

        if((sizeof desktopGroup.content) > maxImages)                   P
        {   FadeTransition                                              P
            {   node: desktopGroup.content[0];                          P
                fromValue:1;  toValue:0;                                P
                duration: 2s;                                           P
                action: function()                                      P
                {   delete desktopGroup.content[0];                     P
                }                                                       P
            }.play();                                                   P
        }                                                               P
    };
};
// ** Part three below
```

G Thumbnail bar
H Position and size
I Flickr details
J Thumbnail clicked
K Preform transitions together
L Move: onto desktop from thumb bar
M Scale: quarter to full
N Rotate: Full 360, plus/minus random
O Add photo to desktopGroup
P Too many photos on desktop?

Here's a really meaty piece of code for us to get stuck into. The `action` event does all manner of clever things with transitions to spin the photo from the thumbnail bar, onto the desktop. But let's quickly look at the other definitions first. We need the thumbnail bar to sit along the bottom of the screen, so the `width` and `translateY` are bound to the `Scene` object's size. We then plug in the familiar Flickr `apiKey` and `userID` (remember to provide your own key).

The `action` function runs when someone clicks a thumbnail in the `GalleryView`. It passes us the `FlickrPhoto` object, a copy of the thumbnail sized image (we'll use this as a stand-in while the full sized image loads), and the coordinates within the bar of the image, so we can calculate where to start the movement transition.

We need a photo to work from, and the `createPhoto()` is a handy private function that creates one for us, complete with white border, shadow, loading progress bar, and the ability to be dragged around the desktop. We'll examine its code at the end of the listing; for now just accept that `pv` is a new photo to be added to the desktop. The `endX` and `endY` variables are the destination coordinates on the desktop where the image will land. We use the photo's dimensions, from `layoutBounds`, to make it centered.

The `ParallelTransition` is another type of animation transition, except it doesn't actually have any direct effect itself. It acts as a group, playing several other transitions at the same time. In our code we use the `ParallelTransition` with three other transitions, represented in figure 7.9.

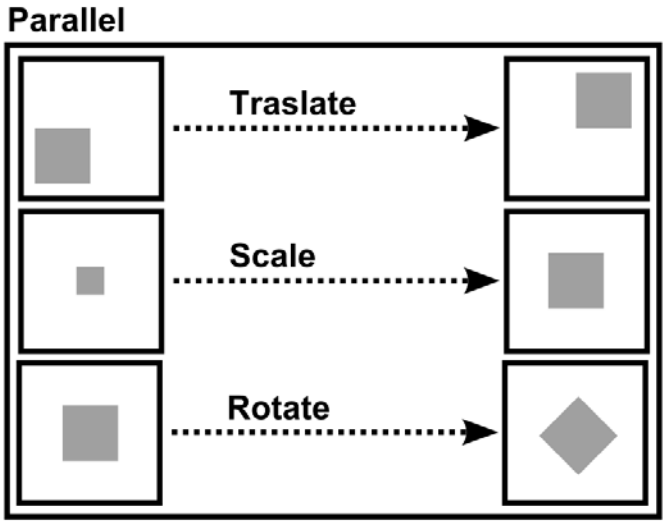


Figure 7.9 All three transitions, translate (movement), scale and rotate, are performed at the same time, to the same scene graph node.

The `TranslateTransition` you've already seen; it moves the image from the thumbnail bar, into the center of the desktop. At the same time the `ScaleTransition` makes it grow from a quarter of its size, to full size, over the same period of time. And finally the `RotateTransition` spins the node around by 360 degrees, plus or minus a random angle, but no more than 30 degrees either way. Note how we only have to specify the actually target node at the `ParallelTransition` level, not at every transition contained within it.

We fire off the transition to run in the background, then add the new photo to the `desktopGroup`, to ensure it's displayed on the screen.

The final block of code uses yet another transition; this time a `FadeTransition`. We don't want too many photos on the desktop at once, so once the maximum has been exceeded we kick off a fade to make the oldest photo gracefully vanish, then (using an `action`) we delete it from the `desktopGroup`.

In the third part of the `PhotoViewer` code (listing 7.19)we'll finish of the loose ends, so we can build the scene graph.

Listing 7.19 PhotoViewer.fx (part 3)

```
// ** Part two above
```

```
def controls:VBox = VBox                                                    Q
{   translateX: bind sc.width - buttonW;
    translateY: bind sc.height -
        controls.layoutBounds.height;
    spacing: 5;
    content:
    [   SwingButton                                                         R
        {   text: "Clear";                                                  R
            width: buttonW;  height: buttonH;                               R
            action: function() { desktopGroup.content = []; }               R
        } ,                                                                 R
        SwingButton                                                         S
        {   text: "Next";                                                   S
            width: buttonW;  height: buttonH;                               S
            action: function() { galView.next(); }                          S
        } ,                                                                 S
        SwingButton                                                         T
        {   text: "Previous";                                               T
            width: buttonW;  height: buttonH;                               T
            action: function() { galView.previous(); }                      T
        }                                                                   T
    ],
}

var vb:VBox;
Stage
{   scene: sc = Scene
    {   content:                                                            U
        [   mouseHandler, desktopGroup,                                     U
            galView, controls, exitButton                                   U
        ];                                                                  U
        fill: LinearGradient
        {   endX: 1;  endY: 1;  proportional: true;
            stops:
            [   Stop { offset: 0;  color: Color.WHITE; } ,
                Stop { offset: 1;  color: Color.GOLDENROD; }
            ];
        };
        stylesheets: [ "{__DIR__}PV.css" ]                                  V
    }
    fullScreen: true;                                                       W
};
// ** Part four below
```

Q Clear/prev/next buttons
R Clear: empty desktopGroup
S Next: next page
T Previous: last page
U Add to application scene
V Plug stylesheet in
W Full screen, please

Listing 7.19 is quite tame compared to the previous two parts of `PhotoViewer`. Three Swing buttons allow the user to navigate the gallery using the exposed function in `GalleryView`, and to clear the desktop by emptying `desktopGroup.contents`,

WARNING: SWING WILL SOON BE SLUNG

This chapter is already long enough without developing a custom button (as we did in chapter six). For this reason I've used the Swing wrappers for the handful of GUI widgets we need. Be warned, however: the Swing library is being pushed firmly into the background once JavaFX gets its own controls API. If you have a spare ten minutes, why not write your own custom button class for this project, and make it look *really* cool?

Listing 7.20 is the final piece of code in this source file, and indeed the entire project. Whew!

Listing 7.20 PhotoViewer.fx (part 4)

```
// ** Part three above
function createPhoto(photo:FlickrPhoto, image:Image) : Node
{   var im:Image;
    var iv:ImageView;
    var pr:Progress;
    var x_im:Image = image;  // Local variable                          X

    def w:Number = bind iv.layoutBounds.width + 10;                     Y
    def h:Number = bind iv.layoutBounds.height + 10;                    Y

    def n:Group = Group
    {   var dragOriginX:Number;                                         Z
        var dragOriginY:Number;                                         Z
        content:
        [   Rectangle                                                  AA
            {   translateX: 15;  translateY: 15;                       AA
                width: bind w;  height: bind h;                        AA
                opacity: 0.25;                                         AA
            } ,                                                        AA
            Rectangle                                                  BB
            {   width: bind w;  height: bind h;                        BB
                fill: Color.WHITE;                                     BB
            } ,                                                        BB

            iv = ImageView                                             CC
            {   translateX: 5;  translateY: 5;                         CC
                fitWidth: FlickrPhoto.MEDIUM;                          CC
                fitHeight: FlickrPhoto.MEDIUM;                         CC
                preserveRatio: true;                                   CC
                //smooth:true;                                         CC
                image: im = Image                                      CC
                {   url: photo.urlMedium;                              CC
                    backgroundLoading: true;                           CC
                    placeholder: image;                                CC
                };                                                     CC
            } ,                                                        CC
            pr = Progress                                              DD
            {   translateX: 10;  translateY: 10;                       DD
                minimum:0;  maximum:100;                               DD
                value:  bind im.progress;                              DD
                visible: bind (pr.value<pr.maximum);                   DD
            }                                                          DD
        ];
```

```
            blocksMouse: true;                                              EE
            onMousePressed: function(ev:MouseEvent)
            {   dragOriginX=ev.x;  dragOriginY=ev.y;                        FF
            }
            onMouseDragged: function(ev:MouseEvent)
            {   n.translateX += ev.x-dragOriginX;                           GG
                n.translateY += ev.y-dragOriginY;                           GG
            }
        };
    }
```

X Workaround for Java limitation
Y Photo size (without shadow)
Z Drag variables
AA Shadow rectangle
BB White border rectangle
CC ImageView displays thumb or photo
DD Progress bar bound to loading
EE Mouse events stop here
FF Record click start coords
GG Update coordinates from drag

We've saved the best to last; this is real heavy-duty JavaFX Script coding here! The `createPhoto()` function constructs a scene graph node to act as our full sized desktop photo, but it does more than just that.

- It creates a white border and shadow to fit around the photo, matched to the correct dimensions of the image.
- It displays a scaled thumbnail, and kicks off the loading of the full sized image.
- It displays a progress meter while we wait for the full image to load.
- It allows itself to be dragged by the mouse, within its parent (the desktop).

At the top of the function we define some variables to reference parts of the scene graph we're creating. The `x_im` is interesting, it's actually a work around for a limitation caused by the Java Virtual Machine, and the way it handles something called anonymous inner classes. I won't bore you with the details, mainly because they require a detailed knowledge of how variables in different *scopes* interact inside the JVM; suffice to say in rare circumstances the JavaFX Script compiler may complain a given variable needs to be made “final” (Java's way of saying immutable). The work around is to make a local reference to said object, as we see with `x_im`.

Back to the code: the variables `w` and `h` are the size of the photo, including its white border. The first `Rectangle` inside our `Group` is the shadow, using the default color (black) set to a quarter opacity. I experimented with using the `DropShadow` effect class inside `javafx.scene.effect`, but found it affected the application's frame rate too much, so this `Rectangle` is a kind of *poor man's* alternative. The shadow is followed by the white photo border, and this in turn is followed by the `ImageView` to display our photo.

We make sure the image is sized to fit the proportions of Flickr's medium sized photo,and we point to the medium photo's URL from the `FlickrPhoto` object. Here's the clever part:

we assign the thumbnail as the placeholder while we wait for the full sized image to load. This ensures we get *something* to display immediately, even if it's blocky.

The progress bar completes the scene graph contents. It will only be visible so long as there's still some loading left to be done.

That's the scene graph, now it's time to look at the event handlers.

```
BlocksMouse: true;
onMousePressed: function(ev:MouseEvent)
{   dragOriginX=ev.x;  dragOriginY=ev.y;
}
onMouseDragged: function(ev:MouseEvent)
{   n.translateX += ev.x-dragOriginX;
    n.translateY += ev.y-dragOriginY;
}
```

Reproduced above is the chunk of code responsible for allowing us to drag the node around the desktop display. The `blocksMouse` setting is very important, without it we'd get all manner of bizarre effects whenever we moved a photo. When the mouse button goes down on a given part of the screen there may be numerous scene graph nodes layered up underneath it; who gets the event? The highest node? The lowest node? All of them?

The answer is “all of them”, working from front to back, unless we take action to stop it! If we allowed mouse events to be applied to both a photo and the desktop, we'd get two sets of actions at once. Sometimes this is desirable, but in our case it is not–the event should either go to an individual photo, or the desktop, but not both. The `blocksMouse` property prevents mouse events from going any further down the list of nodes, once this node has seen them.

The remainder of the event code should be fairly obvious. When the mouse goes down we record its start position, relative to the top left corner of the node. Even though the node will be rotated on the desktop, the mouse coordinates still work in sync with the rectangular shape of the photo, because the coordinates are local to the interior of the node (see figure 7.10). Drag events then update the translation of the node within its parent, causing it to move inside the desktop.

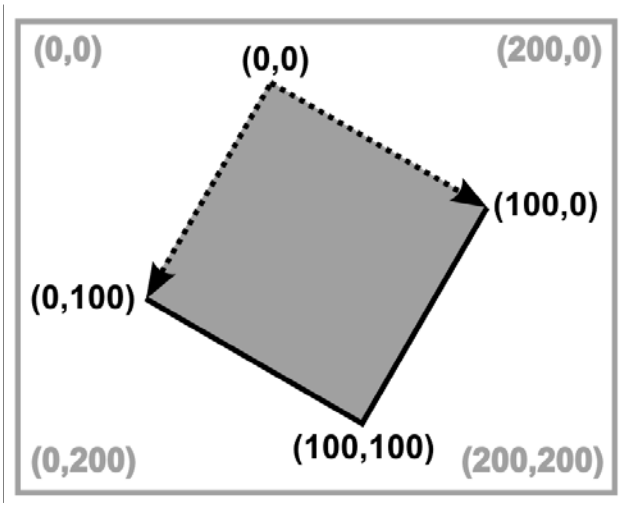


Figure 7.10 The coordinate system of a node always matches the rotation of the node itself. The gray rectangle has been rotated 30 degrees clockwise, yet its local coordinate system is unaffected.

Now our JavaFX Script code is done and dusted, we'll round off the application with its stylesheet.

7.4.4 Styling our application's Progress control

The main application class, above, made use of its own CSS file to style the `Progress` control. The file was called `PV.css`, and you'll find it (if you downloaded the source code from the project's web site) in the project's `res` directory. Before you run the PhotoViewer application, `PV.css` (listing 7.21, below) needs to be copied into the `jfxia.chapter7` package inside the build directory.

Listing 7.21 PV.css

```
Progress
{   boxWidth: 10;
    boxHeight: 20;
    boxCount: 20;
    boxSetStroke: Peru;
    boxSetFill: linear (0%,0%) to (0%,100%) stops
        (0.0,Gold), (0.25,White), (0.50,Gold), (0.85,Peru);
    boxUnsetStroke: Gold;
    boxUnsetFill: White;
}
```

And that, as they say, is that! We now have a fully functional Flickr photo viewer.

7.5 Size matters: node bounds in different contexts

In this project we had a lot of rotating and scaling of nodes, and we saw prominent use of `layoutBounds` and other node properties. As you browsed the JavaFX API documentation you may have noticed every node has several `Rectangle2D` properties, publicizing its location and size. You may be wondering why there are so many (wouldn't one be enough?) and how they relate to each other.

The truth is they detail the size of the node at different points during its journey from abstract shape in a scene graph, to pixels on the video screen. (Remember, the `javafx.geometry.Rectangle2D` class defines two x and y coordinates, one for the top-left corner, and the other for the bottom-right. It does **not** use x,y/width,height.) The journey looks like this:

1. We start with the basic node.
2. Next we perform any effects (reflections, drop shadow, etc.), as specified by the node's `effect` property.
3. If the `cache` property is set to true, the node may be cached as a bitmap to speed future updates.
4. Opacity is applied next, as per the `opacity` property.
5. The node is clipped according to its `clip` property, if set.
6. **At this point `boundsInLocal` is calculated.**
7. Any `transforms` are then applied, translating, rotating and scaling the node.
8. **At this point `layoutBounds` is calculated.**
9. The node is scaled, by `scaleX` and `scaleY`.
10. The `rotate` property is now applied.
11. The node origin is moved within its parent, using `translateX` and `translateY`.
12. **At this point `boundsInParent` is calculated.**

In layman's terms this means `boundsInLocal` is always the position and size as the node sees itself, from it's local, internal, perspective. It ignores any translations, scaling or rotations which may have been applied. This is the coordinate space that the node's own variables (x, y, width, height, radius, whatever) work in, and also mouse events are delivered to the node in.

The `layoutBounds` takes account of transformations applied to the node, and can be used to position it relative to sibling nodes in the scene graph. It can also be used by a layout container to constrain the node, which is why it is `public-init` rather than `public-read`.

The `boundsInParent` is the true location and size of the node, once all operations have been applied, as seen from the perspective of it's parent.

Further reading

Senior Staff Engineer at Sun Microsystems, Amy Fowler, has written an extensive blog entry on how layout works in JavaFX, going into far more detail than I could ever hope to cover here. Check it out at the following URL (broken over two lines):

```
http://weblogs.java.net/blog/aim/archive/
  2009/01/layout_primer_f_1.html
```

7.6 Summary

This has been quite a long project, and along the way we've dealt with a lot of important topics. We started by looking at addressing a web service and parsing the XML data it gave back. This is a vital skill, as the promise of *cloud computing* is ever more reliance on web services as a means of linking software components. Although we only addressed one Flickr method, the technique is the same for all web service calls. You should be able to extend the classes in the first part of our project to talk to other bits of the Flickr API with ease. Why not give it a try?

In the second part of the project we saw how to create controls which can be manipulated by CSS-like stylesheets. In future versions of JavaFX this will be an essential tool, closing the gap between designer and coder. Unfortunately, given the current state of the v1.1 SDK, I was only able to give a taste of how skinning will work, but I have it on good authority the finished article will not be a million miles away from the details listed in this book.

We ended by throwing nodes around the screen with reckless abandon, thanks to our new friend the transition. Transitions take the sting out of creating beautiful UI animation, and as such deserve prime place in any JavaFX programmer's toolbox. Why not experiment with the other types of transition `javafx.animation.transition` has to offer?

You'll be glad to know not only do we now have a nice little application to view our own (or someone else's) photos with, but we've passed an important milestone on our journey to master JavaFX. With this chapter we've now touched on most of the important scene graph topics. Sure, we didn't get to use *every* transition, or try out *every* different effect, we didn't even get to play with *every* different type of shape in the `javafx.scene.shape` package, but we've covered enough of a representative sample that you should now be able to find your way around the JavaFX API documentation without getting hopelessly lost.

In the next chapter we'll move away from purely language and API concerns, to look at how designers and programmers can work together, and how to turn our applications into applets. Until then, I encourage you make this project's code your own–change those boring Swing buttons for cool styled ones, add a text field for the ID of the gallery to view, and have a lot more fun with transitions. Whatever you do, from now on, do it with style!

8

Web coding

Previous chapters have concerned themselves with language syntax and programming APIs; the nuts and bolts of coding JavaFX. This chapter we're focusing more on how JavaFX fits into the wider world, both at development time and at deployment time.

One of the goals of JavaFX was to recognize the prominent role graphic artists and designers play in modern desktop software, and bring them into the application development process in a more meaningful way. To this end Sun created the JavaFX Production Suite, a collection of tools for translating content from the likes of Adobe Photoshop and Illustrator into a format manipulable by a JavaFX program.

Another key goal of JavaFX was to provide a single development technology, adaptable to many different environments (as mentioned in the introductory chapter). The 1.0 release concentrated on getting the desktop and web right, phones followed with 1.1 in February 2009, and the TV platform is expected sometime in 2009/10. The snappily titled "*release 6 update 10"* of the Java Runtime Environment offers an enhanced experience for desktop applications and web applets. The new features mainly center around making it less painful for end users to install updates to the JRE, easier for developers to ensure the right version of Java is available on the end user's computer, and considerably easier for the end user to rip their favorite applet out of the browser (literally) and plonk it down on the desktop as a stand alone application.

In this chapter we're going to explore both these ideas. We'll have a little fun developing a JavaFX application, then transform it into an applet to run inside a web browser. We'll also take some SVG (Scalable Vector Graphics) images and translate them so they can be plugged directly in a JFX scene graph.

As with previous chapters, we'll be learning by example. So far our projects have been entertainment based–it's about time we did something more practical. More and more serious applications are getting glossy user interfaces, and you don't get more serious than when the output from your program could change the fate of nations.

8.1 The Enigma project

These days practically everyone in the developed world has used encryption. From on-line shopping to cell phone calls, encryption is the oil which lubricates modern digital networks. It's interesting to consider, then, that machine based encryption (as opposed to the pencil and paper variety) is a relatively recent innovation. Indeed not until the Second World War did automated encryption achieved wide spread adoption, with one technology in particular becoming the stuff of legend.

The Enigma Machine was an electro-mechanical device, first created for commercial use in the early 1920, but soon adopted by the German military for scrambling radio Morse Code messages between commanders and their troops in the field. With its complex every-shifting encoding patterns, the Enigma's messages were believed to be unbreakable; presumably by the same people who thought the Titanic was unsinkable! As it happens the Enigma cipher *was* broken, not once, but twice!

Creating our own Enigma Machine will give us a practical program with a *real world* purpose (albeit one sixty years old). Having reconstructed our own little piece of computing history, we'll be turning it from a desktop application into an applet, and then letting the end user turn it back into a desktop application again with the drag and drop of a mouse.

8.1.1 The mechanics of the Enigma cipher

Despite its awesome power, the Enigma Machine was blissfully simple by design. Pre-dating the modern electronic computer, it relied on old fashioned electric wiring and mechanical gears. The system consisted of four parts: three *rotors*, and a *reflector*.

Each rotor is a disk with fifty-two electrical contacts, twenty-six on one face and another twenty-six on the other. Inside the disk each contact is wired to another on the opposite face. Electric current entering at position one might exit on the other side in position eight, for example. The output from one rotor was passed into the next through the contacts on the faces, forming a circuit.

In figure 8.1 current entering Rotor A in the position 1 leaves in position 8. Position 8 in the next rotor is wired to position 3, and 3 in the last rotor is wired to 22. The final component, the reflector, directs the current back through the rotors in the reverse direction. This ensures pressing A will give W, and (more importantly, when it it comes time to decode) pressing W will give A.

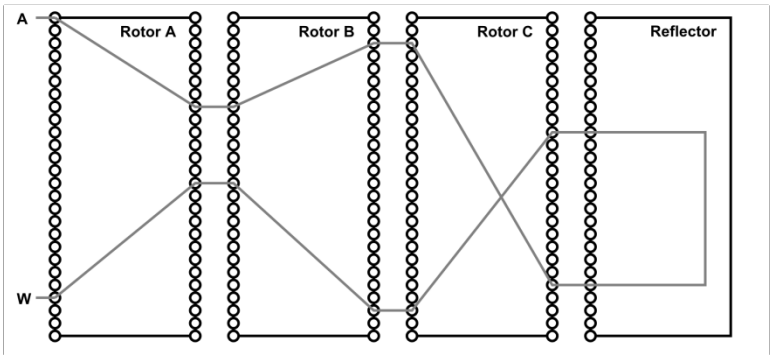


Figure 8.1 Tracing one path inside the Enigma, forming a circuit linking key A with lamp W, and key W with lamp A. But as the rotors move, the circuit changes to provide a different link.

Each time a letter is encoded, one or more rotors turn, changing the combined circuit. This constant shifting of the circuit is what gives the Enigma part of its power; each letter is encoded differently with successive presses. To decode a message successfully one must set the three rotor disks to the same start position as when the message was originally encoded. Failure to do so results in garbage.

8.2 The Enigma Machine, version 1

The first stab we're going to have at an Enigma Machine will just put the basic encryption, input and output components in place. In this initial version of the project we're stick to familiar ground by developing a desktop application; the web applet will come later. You can see what the application will look like from figure 8.2. In the next version we'll flesh it out with a nicer interface and better features.

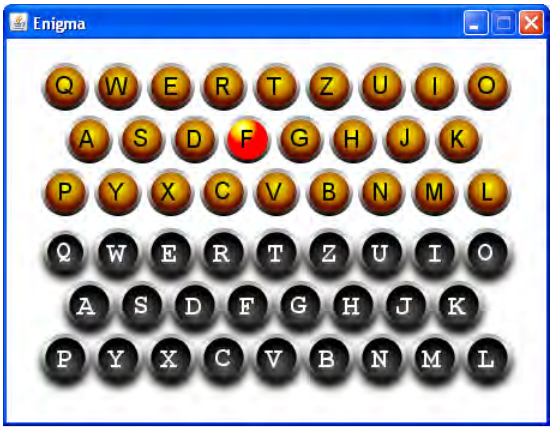


Figure 8.2 Our initial version of the Enigma Machine will just provide basic encryption, input and output, served up with a splash of visual flare, naturally!

To create the key and lamp graphics we'll be using two SVG (Scalable Vector Graphics) files, which we'll translate to JavaFX's own FXZ file format using the tools in the JavaFX Production Suite. In keeping with this book's policy of not favoring one platform, or tool, or IDE, I used the Open Source application Inkscape to draw the graphics, and the SVG converter tool to turn them into JavaFX scene graph compatible files. If you are lucky enough to own Adobe Photoshop CS3 or Illustrator CS3, plugins are available in the suite to export directly into JavaFX's FXZ format from inside those programs.

The SVG / IDE-agnostic route isn't that different to working with the Adobe tools and using the NetBeans JFX plugin. For completeness I'll give a quick rundown of the Adobe / NetBeans shortcuts at the end of the first version.

8.2.1 Getting ready to use the JavaFX Production Suite

To join in with this chapter you'll need to download a few things. The first is the Java Production Suite, available from the official JavaFX site; download this and install it. The JavaFX Production Suite is a separate download to the SDK because the suite is intended for use by designers, not programmers. But, strangely, it contains the JAR files and documentation needed by programmers to use a designer's creations in JavaFX code!

> **IMPORTANT: JAVAFX 1.0 USERS, CHECK YOUR CLASSPATH**
>
> For code in this project to compile or run under the old JavaFX 1.0 release (December 2008) the `javafx-fxd-1.0.jar` file shipped with the JavaFX Production Suite must be on the classpath. Strangely the classes for reading FXZ files were not included in the JavaFX SDK itself until JavaFX 1.1 (February 2009).
>
> The JAR file lives in the the `Libraries` directory, inside your JavaFX Production Suite installation. The `README.html` in the root directory of the installation contains further details, as well as a link to the library's Javadocs.

The second download is the project source code, including the project's SVG and FXZ files, available from this book's web page. Without this you won't have the graphics files to use in the project.

The final download is entirely optional: Inkscape, an Open Source vector graphics application, which you can use to draw SVG files yourself. As the SVG files are already available to you in the source code download, you won't need Inkscape for this project, but you may find it useful when developing your own graphics. As noted, you could alternatively use Photoshop CS3 (for bitmap images) or Illustrator CS3 (for vector images), if you have access to them.

The links

```
http://javafx.com/  (follow the download link)
http://www.manning.com/morris/
http://www.inkscape.org/
```

8.2.2 Converting SVG files to FXZ

If you downloaded the source code you should have access to the SVG files used to create the images in this part of the project. Scalable Vector Graphics is a World Wide Web Consortium (W3C) standard for vector images on the web. Vector images, unlike their bitmap cousins, are defined in terms of geometric points and shapes, making them scalable to any display resolution. This fits in nicely with the way JavaFX's scene graph works.

There are two files supplied with the project, `key.svg` and `lamp.svg`, defining the vector images we'll need. They're in the `svg` directory of the project. Figure 8.3 shows the key image being edited by Inkscape. To bring these images into our JavaFX project we need to translate them from their SVG format into something closer to JavaFX Script. Fortunately for us the JavaFX Production Suite comes with a tool to do just that.

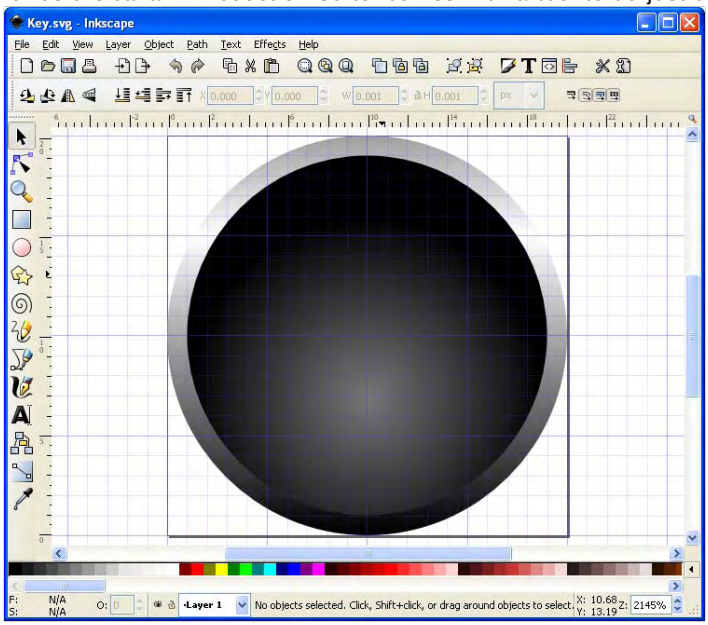


Figure 8.3 SVG images are formed from a collection of shapes; for the key just two circles with gradient fills JavaFX also supports layered bitmaps from Photoshop, and vector images from Illustrator.

The format JavaFX uses is called FXZ (at least, that's the three letter extension its files carry), which is a simple Zip file holding one or more components. Chief amongst these is

the FXD file, a JavaFX Script scene graph declaration. There may also be other supporting files, such as bitmap images or fonts. (If you want to poke around inside a FXZ, just change its file extension from `.fxz` to `.zip`, and it should open in your favorite Zip application.)

For each of our SVG files the converter parses the SVG and outputs the equivalent JavaFX Script code, which it stores in the FXD file. Any supporting files (for example, texture bitmaps) are also stored, and the whole thing is zipped up to form a FXZ file.

Once the JavaFX Production Suite is installed the SVG converter should be available as a regular desktop application (for example, via the Start Menu on Windows). You can see what it looks like in figure 8.4.

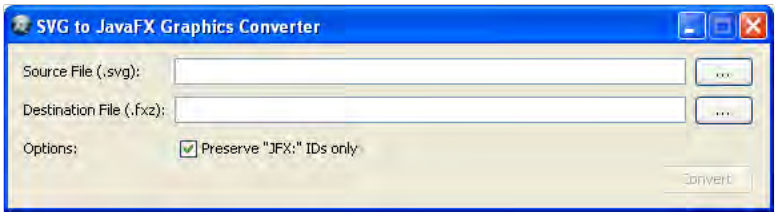


Figure 8.4 The SVG Converter takes SVG files, using the W3C's vector format, and translates them into FXZ files, using JavaFX's declarative scene graph markup.

To run the converter we merely need to provide the filenames of the source SVG, and the destination FXZ. We can also choose whether to retain "`jfx:`" IDs during the conversion, which demands some extra explanation.

While creating the original SVG file (and this is also true of Illustrator and Photoshop files too) it is possible to mark particular elements and expose them to JavaFX after the conversion. We do this by assigning the prefix "`jfx:`" (that's JFX followed by a colon) to the layer or element's ID. When the conversion tool sees that prefix on an ID, it stores a reference to the element in the FXD, allowing us to later address that specific part of the scene graph in our own code. Later one we'll see an example of doing just that, when we play with the lamp graphic. Generally you should ensure the option is switched on when running the tool.

TIP: CHECK YOUR ID

When I first attempted to use FXZ files in my JavaFX programs, the FXD reader didn't seem to find the parts of the image I'd carefully labeled with the "`jfx:`" prefix. After a good fifteen minutes of frustration, I realized my schoolboy error: naming the layers of a SVG is not the same as setting their IDs. The JavaFX Production Suite relies on the ID only. So if, like me, you experience trouble finding parts of your image once it has been loaded into JavaFX, double check the IDs on your original SVG.

The JavaFX Production Suite also comes with a utility to display FXZ files, called the JavaFX Graphics Viewer. After generating your FXZ you can use it to check the output. If

you haven't done so already, try running the converter tool, and generating FXZ files from the project's SVGs, then use the viewer to check the results.

We'll use the resulting FXZ files when we develop the lamp and key classes, a little later. Right now you need to be aware the FXZ files should be copied into the `jfxia.chapter8` package of the build, so they live next to the two classes which load them, otherwise the application will fail when you run it.

8.2.3 The Rotor class: the heart of the encryption

The `Rotor` class is the heart of the actual encryption code. Each instance models a single rotor in the Enigma. Its twenty-six positions are labeled A to Z, but should not be confused with the actual letters being encoded or decoded. The assigning of a letter for each position is purely practical; operators needed to encode rotor start positions into each message, and the machine had no digit keys. However, for convenience, we'll configure each rotor using the letter corresponding to each position. As the current can pass in either direction through the rotor wiring, we'll build two lookup tables, one for left to right, and one for right to left. Listing 8.1 has the code.

Listing 8.1 Rotor.fx (version 1)

```
package jfxia.chapter8;

package class Rotor
{   public-init var wiring:String;
    public-init var turnover:String;
    public-read var rotorPosition:Integer = 0;
    public-read var isTurnoverPosition:Boolean = bind
        (rotorPosition == turnoverPosition);

    var rightToLeft:Integer[];
    var leftToRight:Integer[];
    var turnoverPosition:Integer;

    init
    {   rightToLeft = for(a in [0..<26]) { -1; }                          A
        leftToRight = for(a in [0..<26]) { -1; }                          A
        var i=0;                                                          A
        while(i<26)                                                       A
        {   var j:Integer = chrToPos(wiring,i);                           A
            rightToLeft[i]=j;                                             A
            leftToRight[j]=i;                                             A
            i++;                                                          A
        }                                                                 A

        if(isInitialized(turnover))                                       B
            turnoverPosition = chrToPos(turnover,0);                      B
    }

    package function encode(i:Integer,leftwards:Boolean) : Integer        C
    {   var res = (i+rotorPosition) mod 26;                               C
        var r:Integer = if(leftwards) rightToLeft[res]                    C
            else leftToRight[res];                                        C
```

```
        return (r-rotorPosition+26) mod 26;                              C
    }                                                                    C
    package function nextPosition() : Boolean                            D
    {   rotorPosition = if(rotorPosition==25) 0                          D
            else rotorPosition+1;                                        D
        return isTurnoverPosition;                                       D
    }                                                                    D
}

package function posToChr(i:Integer) : String                            E
{   var c:Character = (i+65) as Character;                               E
    return c.toString();                                                 E
}                                                                        E
package function chrToPos(s:String,i:Integer) : Integer                  E
{   var ch:Integer = s.charAt(i);                                        E
    return ch-65;                                                        E
}                                                                        E
```

A Wiring connections set as string
B When should next rotor move?
C Encode an input
D Step to next position, returning turnover
E Convert between letter and position

Looking at listing 8.1 we can see the wiring is set using a `String`, which makes it easy to declaratively create a new rotor object. Other public properties control how the encryption works and how the rotors turn.

- The variable `wiring` is the source for two lookup tables: `rightToLeft` gives the outputs for positions A to Z (0 to 25) on one face, and for convenience `leftToRight` does the reverse path from the other face. If the first letter in `wiring` was D, for example, the first entry of table 1 would be 3 (D), and the fourth entry of table 2 would be 0 (A).
- The private sequences `rightToLeft` and `leftToRight` detail the outputs for each input position, zero to twenty-five, in either direction. They are set based on `wiring`.
- The variable `turnover` is the position (as a letter) at which the next rotor should step. The private variable `turnoverPosition` is the `turnover` in its more useful numeric form. One might have expected a rotor to do a full A to Z cycle before the next rotor ticks over one position–but the Enigma designers thought this was too predictable. The `isTurnoverPosition` variable is a handy bound flag for assessing whether this rotor has just entered its turnover position.
- The `rotorPosition` property is the current rotation offset of this rotor, as an `Integer`. If this is set to 2, for example, then an input for position 23 would actually enter the rotor at position 25.

So that's our `Rotor` class. Each rotor provides one part of the overall encryption mechanism. We can use a `Rotor` object to create the reflector too–we just need to ensure the `wiring` string models symmetrical paths. Figure 8.5 show how this might look.

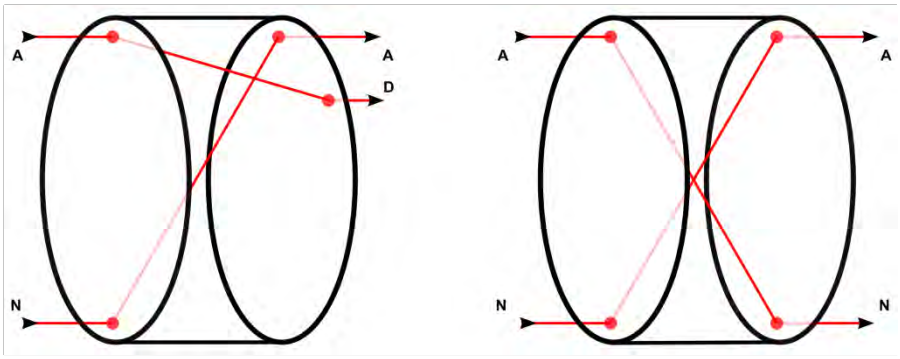


Figure 8.5 Two disks, with left/right faces. The rotor wiring is not symmetrical (left), but we can create a reflector from a rotor by ensuring thirteen of the wires mirror the path of the other thirteen (right).

The regular rotors are only symmetrical by reversing the direction of current flow. Just because N (left input) results in A (right output), does not mean A will result in N when moving in the same left-to-right direction. Figure 8.5 shows this relationship in its left hand rotor. We can, however, conspire to create a rotor in which these connections are deliberately mirrored (see figure 8.5's right hand rotor) and this is precisely how we model the reflector. This convenience saves us from needing to create a specific reflector class.

8.2.4 A quick utility class

Before we proceed with the scene graph classes, we need a quick utility class to help position nodes. Listing 8.2, which does that, is up next.

Listing 8.2 Util.fx

```
package jfxia.chapter8;

import javafx.scene.Node;

package bound function center(a:Node,b:Node,hv:Boolean) : Number
{   var aa:Number = if(hv) a.layoutBounds.width
        else a.layoutBounds.height;
    var bb:Number = if(hv) b.layoutBounds.width
        else b.layoutBounds.height;
    return ((aa-bb) /2);
}
package bound function center(a:Number,b:Node,hv:Boolean) : Number
{   var bb:Number = if(hv) b.layoutBounds.width
        else b.layoutBounds.height;
    return ((a-bb) /2);
}
```

The two functions in listing 8.2 are used to center one node inside another. The first function centers node `b` inside node `a`, the second centers node `b` inside a given dimension.

In both cases the boolean hv controls whether the result is based on the width (true) or the height (false) of the parameter nodes.

8.2.5 The Key class: input to the machine

The real Enigma machine used keys and lamps to capture input and show output. To remain faithful to the original we'll create our own Key and Lamp custom nodes. The Key is first; see listing 8.3.

Listing 8.3 Key.fx

```
package jfxia.chapter8;

import javafx.fxd.FXDLoader;
import javafx.scene.Node;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.paint.Color;
import javafx.scene.input.MouseEvent;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.scene.text.TextOrigin;

package class Key extends CustomNode
{   package def diameter:Number = 40;
    def fontSize:Integer = 24;

    public-init var letter:String on replace                          A
    {   letterValue = letter.charAt(0)-65;                            A
    }                                                                 A
    package var action:function(:Integer,:Boolean):Void;

    def scale:Number = bind if(pressed) 0.9 else 1.0;                 B
    def letterFont:Font = Font.font
    (   "Courier", FontWeight.BOLD, fontSize
    );
    var letterValue:Integer;


    override function create() : Node
    {   def keyNode = FXDLoader                                       C
            .loadContent("{__DIR__}key.fxz")._root;                   C

        keyNode.onMousePressed = function(ev:MouseEvent)              D
        {   if(action!=null)                                          D
                action(letterValue,true);                             D
        };                                                            D
        keyNode.onMouseReleased = function(ev:MouseEvent)             D
        {   if(action!=null)                                          D
                action(letterValue,false);                            D
        };                                                            D

        Group
        {   var k:Node;
            var t:Node;
```

```
            content:
            [   k = keyNode ,                                                E
                t = Text                                                     F
                {   translateX: bind Util.center(k,t,true);                  F
                    translateY: bind Util.center(k,t,false);                 F
                    fill: Color.WHITE;                                       F
                    content: letter;                                         F
                    font: letterFont;                                        F
                    textOrigin: TextOrigin.TOP;                              F
                }                                                            F
            ];
            scaleX: bind scale;
            scaleY: bind scale;
        }
    }
}
```

A Key's letter
B Animation scale
C Load FXZ file into node
D Assign mouse handlers
E FXD node
F Key letter, centered

The `Key` class is used to display an old fashioned looking manual typewriter key on the screen. Its variables are as follows:

- The variable `diameter` and `fontSize` set the size of the key and its key font.
- The variable `letter` is the character to display on this key.
- The function, `action`, is the event callback by which the outside world can learn when our key is pressed or released.
- The private variable `scale` is bound to the inherited variable, `pressed`. It resizes our key whenever the mouse button is down.
- And finally, `letterFont` is the font we use for the key symbol.

The overridden `create()` function is, as always, where the scene graph is assembled. It starts with an unfamiliar couple of lines, reproduced below.

```
def keyNode = FXDLoader
    .loadContent("{__DIR__}key.fxz")._root;
```

This magic incantation loads the FXZ file we created previously, and returns it as a scene graph node. This isn't actually as complex as sounds, given the SVG converter tool has already done all the heavy lifting of converting the SVG format into declarative JavaFX Script code. (Remember: the FXZ file needs to be copied into the build directory for the `jfxia.chapter8` package, or the above code cannot find it!)

```
keyNode.onMousePressed = function(ev:MouseEvent)
{   if(action!=null)
        action(letterValue,true);
};
keyNode.onMouseReleased = function(ev:MouseEvent)
```

```
{   if(action!=null)
        action(letterValue,false);
};
```

Once our key image has been loaded as a node we need to assign two mouse event handlers to it. As the object is already defined we can't do this declaratively, so we must revert to plain old procedural code (ala Java) to hook up two anonymous functions. Both functions call the `action()` event handler, informing the outside world whether the key has been pressed or released.

The rest of the scene graph code should be fairly clear. We need to overlay a letter onto the key, and that's what the Text node does. It uses the utility functions we created earlier to center itself inside the key. At the foot of the scene graph the containing `Group` is bound to the `scale` variable, which in turn is bound to the inherited `pressed` state of the node. Whenever the mouse button goes down, the whole key shrinks slightly, as if being pressed.

DON'T FORGET TO COPY THE FXZ FILES

The FXZ files for this project should be located inside the directory representing the `jfxia.chapter8` package, so a reference to `__DIR__` can be used to find them. Once you've built the project code, make sure the FXZs are alongside the project class files.

8.2.6 The Lamp class: output from the machine

We've just developed a stylized input for our emulator, now we need a similar retro-looking output. In the original Enigma Machine the encoded letters were displayed on twenty-six lamps, one of which would light up to display the output as a key was pressed, so that's what we'll develop next, in listing 8.4.

Listing 8.4 Lamp.fx

```
package jfxia.chapter8;

import javafx.fxd.FXDLoader;
import javafx.fxd.FXDContent;
import javafx.scene.Node;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.scene.text.TextOrigin;

package class Lamp extends CustomNode
{   package def diameter:Number = 40;
    def fontSize:Integer = 20;

    public-init var letter:String;
    package var lit:Boolean = false on replace                    A
```

```
        {   if(lampOn!=null)  lampOn.visible = lit;                          A
        }                                                                    A

        def letterFont:Font = Font.font
        (   "Helvetic", FontWeight.REGULAR, fontSize
        );
        var lampOn:Node;                                                     B

        override function create() : Node
        {   def lampContent:FXDContent = FXDLoader                           C
                .loadContent("{__DIR__}lamp.fxz");                           C

            def lampNode = lampContent._root;                                D

            lampOn = lampContent.getNode("lampOn");                          E

            var c:Node;
            var t:Node;
            Group
            {   content:
                [   c = lampNode ,                                           F
                    t = Text                                                 G
                    {   content: letter;                                     G
                        font: letterFont;                                    G
                        textOrigin: TextOrigin.TOP;                          G
                        translateX: bind Util.center(c,t,true);              G
                        translateY: bind Util.center(c,t,false);             G
                    }                                                        G
                ];
            }
        }
    }
```

A Manipulate lamp FXD
B Our FXD node
C Load as FXDContent type
D Top level
E Specific layer
F Use in scene graph
G Letter text, centered

The `Lamp` class is a very simple binary display node, it has no mouse or keyboard input, but is either lit or unlit.. Once again we load a FXZ file, and bring its content into our code via the `FXDContent` class (remember to copy the FXZ into the build directory). As well as extracting the top level node, however, we also get access to one of the image's internal parts: its "`lampOn`" layer.

If you load up the original SVG files you'll find inside the lamp there's a layer with an alternative version of the center part of the image, making it look lit up (see figure 8.6). The layer is set to be invisible, so it doesn't show by default. It has been given the ID "`jfx:lampOn`", causing the converter to record a reference to it in the FXD.

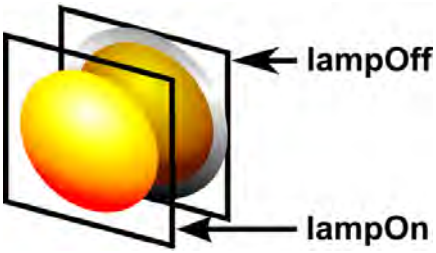


Figure 8.6 The lamp image is constructed from two layers. The lower layer shows the rim of the lamp and its dormant (off) graphic; the upper layer, invisible by default, shows the active (on) graphic.

In the code above we extracted that reference by calling `FXDContent.getNode()` with an ID of "`lampOn`". We don't need the "`jfx:`" prefix with the FXZ/FXD, it was only used in the original SVG to tag the parts of the image we wanted to ID in the FXD. If the ID is found we'll get back a scene graph node, which we store in a variable called `lampOn` (it doesn't *have* to share the same name). We can now manipulate `lampOn` in the code, by switching its visibility on or off whenever the status of `lit` changes.

This ability to design images in third party applications, bring them into our JavaFX programs, then reach inside and manipulable their constituent parts, is very powerful. Imagine, for example, a Chess game where the board and all the various playing pieces are stored in one big FXZ file. To change the graphics the designer need only supply a replacement SVG file, with IDs on key elements that the software expects. The equivalent FXZ file should drop straight into the project as a direct replacement, without need for a rebuild of the code.

8.2.7 The Enigma class: binding the encryption engine to the interface

We've seen three classes so far: the `Rotor`, which provides the basis of our Enigma cipher emulation; the `Key`, which provides the input, and the `Lamp`, which displays the output. Now it's time to pull these classes together with an actual application class, the first part of which is listing 8.5.

Listing 8.5 Enigma.fx (version 1, part 1)

```
package jfxia.chapter8;

import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.effect.DropShadow;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

import java.lang.System;
import java.util.ArrayList;
```

```
def rotors:Rotor[] =                                                    A
[   Rotor { wiring: "JGDQOXUSCAMIFRVTPNEWKBLZYH";                       A
        turnover: "R" } ,                                               A
    Rotor { wiring: "NTZPSFBOKMWRCJDIVLAEYUXHGQ";                       A
        turnover: "F" } ,                                               A
    Rotor { wiring: "JVIUBHTCDYAKEQZPOSGXNRMWFL";                       A
        turnover: "W" }                                                 A
];                                                                      A
def reflector =                                                         A
    Rotor { wiring: "QYHOGNECVPUZTFDJAXWMKISRBL"; }                     A

def row1:String[] = [ "Q","W","E","R","T","Z","U","I","O" ];            B
def row2:String[] = [ "A","S","D","F","G","H","J","K" ];                B
def row3:String[] = [ "P","Y","X","C","V","B","N","M","L" ];            B

def lamps:ArrayList = new ArrayList();
for(i in [0..<26]) { lamps.add(null); }

def innerWidth:Integer = 450;
def innerHeight:Integer = 320;
// Part two below
```

A **Declare the rotors and reflector**

B **Key and lamp layout**

This is just the top part of our application class. The code mainly concerns itself with setting up the three rotors and single reflector, plus defining the keyboard and lamp layout. We'll use the Enigma's authentic keyboard layout; similar to, but not quite the same as, QWERTY. Because we need to manipulate the lamps to display output, we create an array using Java's `ArrayList` collection class of twenty-six elements to hold references to each lamp, in alphabetic order.

Listing 8.6 is the second part of our project, and shows the main scene graph code which builds the window and its contents.

Listing 8.6 Enigma.fx (version 1, part 2)

```
// Part one above
Stage
{   scene: Scene
    {   var l:Node;
        content: Group
        {   content:
            [   l = VBox                                                C
                {   translateX: bind
                        Util.center(innerWidth,l,true);
                    translateY: 20;
                    spacing: 4;
                    content:                                            D
                    [   createRow(row1,false,createLamp) ,
                        createRow(row2,true,createLamp) ,
                        createRow(row3,false,createLamp)
                    ];
                } ,
                VBox                                                    E
                {   translateX: bind l.translateX;
                    translateY: bind l.boundsInParent.maxY+10;
```

```
                    spacing: 4;
                    content:                                                F
                    [   createRow(row1,false,createKey) ,
                        createRow(row2,true,createKey) ,
                        createRow(row3,false,createKey)
                    ];
                    effect: DropShadow                                      G
                    {   offsetX: 0;  offsetY: 5;                            G
                    };                                                      G
                }
            ];
        };
        width: innerWidth;  height: innerHeight;                            H
    }
    title: "Enigma";
    resizable: false;
    onClose: function() { FX.exit(); }
}
// Part three below
```

C Lamp, centered
D Three rows of lamps
E Keys, positioned using lamps
F Three rows of keys
G Drop shadow effect
H Window inner content

The structure is fairly simple, we have two VBox containers (they stack their contents in a vertical alignment, recall), each populated with three calls to `createRow()`. The top VBox is aligned centrally using the utility functions we developed earlier, and 20 pixels from the top of the windows inner bounds. The second VBox has it's X translation bound to its sibling, and its Y translation set to 10 pixels below the bottom of its sibling.

The second VBox has an effect attached to it, `javafx.scene.effect.DropShadow`. We touched on effects previously when we used a reflection in our video player project. Effects manipulate the look of a node; anything from a blur or a color tint, to a reflection or even a pseudo 3D distortion. The `DropShadow` effect merely adds a shadow underneath the node tree it is connected to. This gives our keys a raised 3D effect.

The `createRow()` function manufactures a row of nodes, using a function passed into it (`createLamp` or `createKey`), from a sequence of letters. Its code begins the final part of this source file.

Our `Enigma` class is concluded in listing 8.7, below.

Listing 8.7 Enigma.fx (version 1, part 3)

```
// Part two above
function createRow(row:String[] , indent:Boolean ,                          H
    func:function(l:String):Node) : HBox
{   def size:Integer = 30;
    def h = HBox
    {   spacing: 4;
        content: for(l in [row]) { func(l); };
    };
    if(indent)                                                              I
    {   h.translateX =                                                      I
```

```
            h.content[0].layoutBounds.width/2;                          I
        }                                                               I
        return h;
    }
    function createKey(l:String) : Node                                 J
    {   Key
        {   letter: l;
            action: handleKeyPress;
        };
    }
    function createLamp(l:String) : Node                                K
    {   def i:Integer = l.charAt(0)-65;
        def lamp = Lamp { letter: l; };
        lamps.set(i,lamp);
        return lamp;
    }

    function handleKeyPress(l:Integer,down:Boolean) : Void              L
    {   def res = encodePosition(l);
        (lamps.get(res) as Lamp).lit = down;
        if(not down)                                                    M
        {   var b:Boolean;                                              M
            b=rotors[2].nextPosition();                                 M
            if(b) { b=rotors[1].nextPosition(); }                       M
            if(b) { rotors[0].nextPosition(); }                         M
        }                                                               M
    }

    function encodePosition(i:Integer) : Integer                        N
    {   var res=i;
        for(r in rotors) { res=r.encode(res,false); }                   O
        res=reflector.encode(res,false);                                O
        for(r in reverse rotors) { res=r.encode(res,true); }            O
        return res;
    }
```

H Create rows of lamps or keys
I Intent row, if required
J Function used to create key
K Function used to create lamp
L Handle key up or down
M Key release? Turn rotors
N Encode key position
O Forward then back through rotors

Here we see the `createRow()` function encountered in the previous part. As the layout for both the keyboard and lamp board are the same, a single utility function is employed to position the nodes in each row. The `createKey()` and `createLamp()` functions are passed to `createRow()` for actually making the nodes to lay out. For that important authentic keyboard look, rows can be indented slightly by setting the `indent` boolean.

Finally, we need to consider `handleKeyPress()` and `encodePosition()` functions. The former uses the latter to walk forwards then back across the rotors, with the deflector in the middle, performing each stage of the encryption. Once the input letter is encoded it switches the associated lamp on or off, depending upon the direction the key just moved,

and turns any rotors which need turning (the `nextPosition()` function turns a rotors and returns true if it just hit its *turnover* position).

And so, with the `Enigma` class itself now ready, we have version one of our simulated encryption machine.

8.2.8 Running version 1

What we have with version one is a functional, if not very practical, Enigma emulator. Figure 8.7 shows what to expect when the application is fired up.

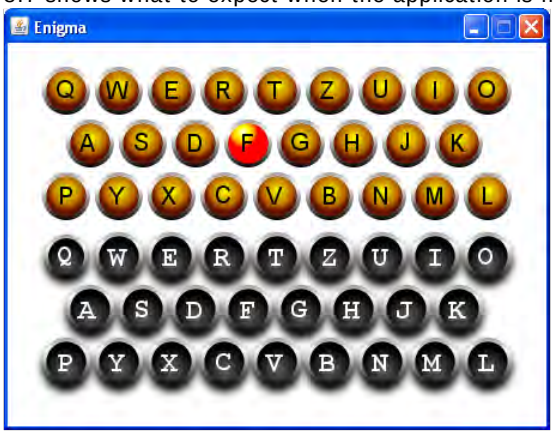


Figure 8.7 This is what we should see when compiling and running version one. It works, but not in a very practical way. The stylized button and lamp nodes help lend the application an authentic feel.

Pressing a key runs its position through the rotors and reflector, with the result displayed on a lamp. There are two clear problems with the current version: firstly, we cannot see or adjust the settings of the rotors, and secondly (in a departure from the real Enigma) we cannot capture the output in a more permanent form.

In the second version of our Enigma we'll be fixing those problems, making the application good enough to put on the web for all to see.

8.2.9 Shortcuts using NetBeans, Photoshop or Illustrator

Just a few quick notes to end this part of the project on. The details in this chapter attempt to be as inclusive as possible. For example, Inkscape was chosen as an example of a SVG editor precisely because it was free and did not favor a particular operating system. However, as mentioned previously, the JavaFX Production Suite does hold some benefits for users of certain well known products. In this section we'll quickly list those benefits.

If you are lucky enough to own *Adobe Photoshop CS3* or *Adobe Illustrator CS3*, or you're working with someone who uses them, the JavaFX Production Suite comes with plugins for

these applications to save directly to FXZ (no need for external tools, like the SVG Converter). Obviously, you need to install the production suite onto the computer running Photoshop or Illustrator, and make sure there's an up to date JRE on there too, but you *don't* need to install the full JavaFX SDK.

Just as we saw with our SVG files, layers and sub-layers in graphics created with these products can be prefixed with "`jfx:`" to preserve them in the FXD definition written into the FXZ file. Fonts, bitmaps, and other supporting data will also be included in the FXZ.

Moving on: the *UI Stub Generator* is a convenience tool for NetBeans users to create JavaFX Script wrappers from FXZ files. Pointing the tool at any FXZ file results in a source code file being generated which extends `javafx.fxd.UiStub`, a subclass of the `FXDContent` class we used above to read the FXZ. Each exposed node in the FXZ (the one's with JFX prefixes in the original image) is given a variable in the class, so it can be accessed easily. An `update()` method is written to do all the heavy lifting of populating the variables with successive calls to `getNode()`. Once generated, NetBeans users can compile this class in their project, and use it in preference to direct `FXDContent` calls.

The UI Stub Generator helps to keep your source files clean from the mechanics of reading FXD data, however once the source code has been generated you may feel the need to edit it. The tool assumes everything is a `Node`, so if you plan on addressing a given part of the scene graph by its true form (for example, a `Text` node), you'll need change the variable's type and add a cast to the relevant `getNode()` line.

All of these tools are fully documented in the help files which come bundled with the JavaFX Production Suite.

8.3 The Enigma Machine, version 2

Version one of the project had a couple of issues which need addressing. First, the user needs to be able to view and change the state of the rotors. Second, some way needs to be introduced to capture the encoded output.

To fix the first of these problems we're going to turn the `Rotor` class into a node which can be used, not only to encode a message, but also set the encryption parameters. To fix the second we'll develop a simple printout display, looking like a teletype page, recording our output along with the lamps. You can see what it looks like at the head of figure 8.8.

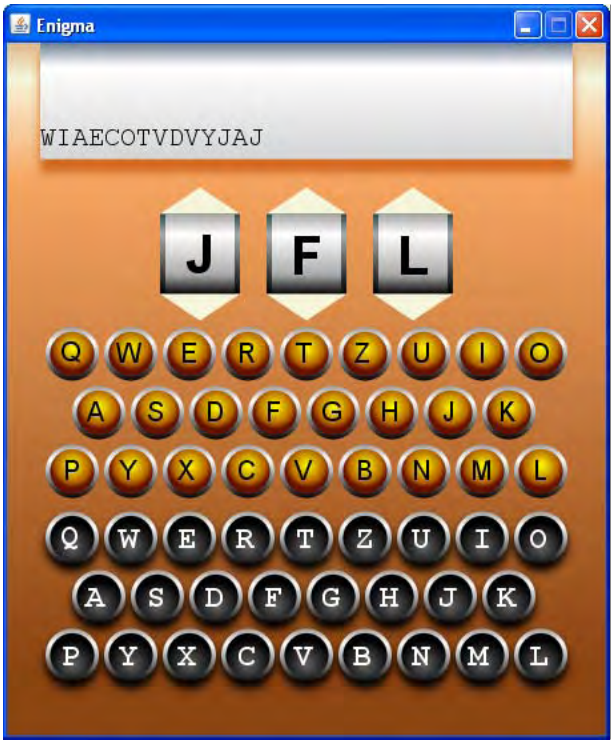


Figure 8.8 Quite an improvement: the Enigma emulator acquires a printout display, and some rotors, as well as an attractive shaded backdrop.

As a quick glance at figure 8.8 betrays, each rotor will display its letter in a shaded display, with arrow buttons constructed from scene graph polygons. The paper node we'll develop will actually wrap over at the top, giving the impression of its text vanishing over a curved surface. Changes to the `Enigma` class will tie the features into the application's interface, giving a more polished look.

8.3.1 The Rotor class, version 2: giving the cipher a visual presence

The `Rotor` class needs a makeover to turn it into a fully fledged custom node, allowing users to interact with it. Because this code is being integrated into the existing class, I've taken the liberty of snipping (omitting for the sake of brevity) the bulk of the code from the previous version, as you'll see from listing 8.8. (It might not save many trees, but it could help save a few branches!) The snipped parts have been marked with bold comments to show what has been dropped, and where. Refer back to listing 8.1 for a reminder of what's missing.

Listing 8.8 Rotor.fx (version 2 – changes only)

```
package jfxia.chapter8;

import javafx.scene.Node;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Stop;
import javafx.scene.shape.Polygon;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.scene.text.TextOrigin;

package class Rotor extends CustomNode
{   // [Snipped variables: see previous version]

    def fontSize:Integer = 40;                                          A
    def width:Number = 60;                                              A
    def height:Number = 60;                                             A
    def buttonHeight:Number = 20.0;                                     A
    def letterFont:Font = Font.font                                     A
    (   "Helvetic" , FontWeight.BOLD, fontSize                          A
    );                                                                  A

    // [Snipped init block: see previous version]

    override function create() : Node
    {   var r:Node;
        var t:Node;
        Group
        {   content:
            [   Polygon                                                 B
                {   points: bind
                    [   width/2 , 0.0 ,
                        0.0 , buttonHeight ,
                        width , buttonHeight
                    ];
                    fill: Color.BEIGE;
                    onMouseClicked: function(ev:MouseEvent)             C
                    {   rotorPosition = if(rotorPosition==0) 25
                            else rotorPosition-1;
                    }
                } ,
                r = Rectangle                                           D
                {   translateY: buttonHeight;
                    width: bind width;
                    height: bind height;
                    fill: LinearGradient
                    {   proportional: true;
                        endX:0;  endY:1;
                        stops:
                        [   Stop { offset: 0.0;
```

```
                        color: Color.DARKSLATEGRAY; } ,
                    Stop { offset: 0.25;
                        color: Color.DARKGRAY; } ,
                    Stop { offset: 1.0;
                        color: Color.BLACK; }
                ]
            }
        } ,
        Rectangle                                                    E
        {   translateX: 4;
            translateY: buttonHeight;
            width: bind width-8;
            height: bind height;
            fill: LinearGradient
            {   proportional: true;
                endX:0;  endY:1;
                stops:
                [   Stop { offset: 0.0;
                        color: Color.DARKGRAY; } ,
                    Stop { offset: 0.25;
                        color: Color.WHITE; } ,
                    Stop { offset: 0.75;
                        color: Color.DARKGRAY; } ,
                    Stop { offset: 1.0;
                        color: Color.BLACK; }
                ]
            }
        } ,
        t = Text                                                     F
        {   translateX: bind Util.center(r,t,true);
            translateY: bind buttonHeight +
                Util.center(r,t,false);
            content: bind posToChr(rotorPosition);
            font: letterFont;
            textOrigin: TextOrigin.TOP;
        } ,
        Polygon                                                      G
        {   translateY: buttonHeight+height;
            points: bind
            [   0.0 , 0.0 ,
                width/2 , buttonHeight ,
                width , 0.0
            ];
            fill: Color.BEIGE;                                       H
            onMouseClicked: function(ev:MouseEvent)
            {   rotorPosition = if(rotorPosition==25) 0
                    else rotorPosition+1;
            }
        }
    ];
  }
}

// [Snipped encode(), nextPosition(): see previous version]
}
// [Snipped posToChr(), chrToPos(): see previous version]
```

A Sizing and font constants

B Up arrow polygon
C Move rotor back one
D Rotor display rim
E Rotor display body
F Rotor current letter
G Down arrow polygon
H Move rotor forward one

The new `Rotor` class contains a hefty set of additions. The scene graph, as assembled by `create()`, brings together several layered elements to form the final image. The graph is structured around a `Group`, at the top and tail of which are `Polygon` shapes.

As its name suggests, the `Polygon` is a node which allows its silhouette to be custom defined from a sequence of co-ordinate pairs. The values in the sequence, `points`, are x then y positions in turn. They form the outline of a shape, with the last co-ordinate connecting back to the first to seal the perimeter. Both of our polygons are simple triangles, with points at the extreme left, extreme right, and in the middle, forming up and down arrows.

The rest of the scene graph consists of familiar components: two shaded rectangles and a text node, forming the rotor body between the arrows.

The arrow polygons have mouse handlers attached to them which change the current `letterPosition`. They ignore the `turnover`, you'll note, as we don't want other rotors flipping over when we're manually adjusting them. As you might expect, the rotor wraps around when it runs past its start or end, hence the condition logic.

So that's the `Rotor` finished. Now for the `Paper` class.

8.3.2 The Paper class: making a permanent output record

The `Paper` class is a neat little display node, with five lines of text which are scaled vertically to create the effect of the lines vanishing over a curved surface. We're employing a fixed width font, for that authentic manual typewriter look. Check out figure 8.9, below. You can almost hear the clack-clack-clack of those levers, hammering out each letter as the keys are pressed.

Figure 8.9 Each line of our Paper node is scaled to create the optical effect of a surface curving away, to accompany the shading of the background Rectangle.

The real Enigma didn't have a paper output. The machines were designed to be carried at the front-line of a battle, and their rotor mechanics and battery were bulky enough without adding a stationery cupboard full of paper and ink ribbons. But we've moved on a little in the eighty plus years since the Enigma was first developed, and while a 1200 dpi laser printer might be stretching credibility a little *too* far, we can at least give our emulator a period teletype display. The code is in listing 8.9, next.

Listing 8.9 Paper.fx

```
package jfxia.chapter8;

import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.effect.DropShadow;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Stop;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.scene.text.TextOrigin;

package class Paper extends CustomNode
{   package var width:Number = 300.0;
    public-read var height:Number;

    def lines:Integer= 5;
    def font:Font = Font.font
    (   "Courier" , FontWeight.REGULAR , 20
    );

    def paper:Text[] = for(i in [0..<lines])                              A
    {   def y:Number = height;                                            B
        def scale:Number = 1.0 – ((lines-1-i)*0.15);                      B
        def lineHeight:Number = 25*scale;                                 B
        height = height+lineHeight;                                       B
        Text
        {   translateY: y;
            font: font;
            textOrigin: TextOrigin.TOP;
            scaleY: scale;
            clip: Rectangle
            {   width: width;  height: lineHeight;
            }
        };
    }

    override function create() : Node
    {   Group
        {   content:
            [   Rectangle                                                 C
```

```
                {   width: bind width;
                    height: bind height;
                    fill: LinearGradient
                    {   proportional: true;
                        endX: 0;  endY: 1;
                        stops:
                        [   Stop { offset:0.0;
                                color: Color.SLATEGRAY; } ,
                            Stop { offset:0.2;
                                color: Color.WHITE; } ,
                            Stop { offset:1.0;
                                color: Color.LIGHTGRAY; }
                        ];
                    }
                    effect: DropShadow
                    {   offsetX: 0;  offsetY: 10;
                        color: Color.CHOCOLATE;
                    };
                } ,
                Group { content: paper; }                                   D
            ];
            onMouseClicked: function(ev:MouseEvent)                         E
            {   add("\n");
            }
        };
    }

    package function add(l:String) : Void                                   F
    {   def z:Integer = lines-1;
        if(l.equals("\n"))
        {   var i:Integer = 1;                                              G
            while(i<lines)                                                  G
            {   paper[i-1].content = paper[i].content;                      G
                i++;                                                        G
            }                                                               G
            paper[z].content="";                                            G
        }
        else
        {   paper[z].content = "{paper[z].content}{l}";                     H
        }
    }
}
```

A Sequence of text lines
B Scale and position to create curve
C Shaded paper backdrop
D Text nodes expanded in place
E Scroll up when clicked
F Add to bottom text line
G Push paper up
H Append to last line

The most interesting thing about the above is the creation of the sequence, `paper`. What this code does is to stack several `Text` nodes atop one another using `translateY`, scaling each in turn from a restricted height at the top, to full height at the bottom. The rather fearsome looking code, `"1.0-((lines-1-i)*0.15)"`, subtracts multiples of 15% from the

height of each line, so the first line is scaled the most, and the last line not at all. The result is a curved look to the printout, just what we wanted!

The rest of the listing is unremarkable scene graph code, apart from the `add()` function at the end. This is what the outside world uses to push new letters onto the bottom line of the teletype display. Normally the letter is appended to the end of the last `Text` node, but if the letter is a carriage return each `Text` node in `content` is copied to its previous sibling, creating the effect of the paper scrolling up a line. The last `Text` node is set to an empty string, ready for fresh output.

We now have all the classes we need for our finished Enigma machine. All that's left is to refine the application class itself to include our new graphical rotors and paper.

8.3.3 The Enigma class, version 2: at last our code is ready to encode

Having upgraded the Rotor class, and introduced a new Paper class, we need to integrate these into the display. Listing 8.10, below, does just that.

Listing 8.10 Enigma.fx (version 2, part 1 – changes only)

```
package jfxia.chapter8;

import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.effect.DropShadow;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Stop;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

import java.lang.System;
import java.util.ArrayList;

// [Snipped defintions for rotors, reflector, row1, row2,
//   row3 and lamps: see previous version]

def innerWidth:Integer = 450;
def innerHeight:Integer = 520;

var paper:Paper;
// Part two below
```

Listing 8.10 is the top part of the updated application class; adding little to what was already there in the previous version. That's why I've snipped parts of the content again, indicated (as before) with comments in bold. You can refresh your memory by glancing back at listing 8.5. As you can see, the window has been made bigger to accommodate the new elements we're about to add, and we've created a variable to hold one of them, `paper`.

The main changes come in the next part, listing 8.11, the scene graph.

Listing 8.11 Enigma.fx (version 2, part2)

```
// Part one above
Stage
{   scene: Scene
    {   content: Group
        {   var p:Node;
            var r:Node;
            var l:Node;
            content:
            [   p = paper = Paper                                          A
                {   width: innerWidth-50;
                    translateX: 25;
                } ,
                r = HBox                                                   B
                {   translateX: bind
                        Util.center(innerWidth,r,true);
                    translateY: bind p.boundsInParent.maxY;
                    spacing: 20;
                    content:
                    [   rotors[0], rotors[1], rotors[2]
                    ];
                } ,
                l = VBox
                {   translateX: bind Util.center(innerWidth,l,true);
                    translateY: bind r.boundsInParent.maxY+5;
                    spacing: 4;
                    content:
                    [   createRow(row1,false,createLamp) ,
                        createRow(row2,true,createLamp) ,
                        createRow(row3,false,createLamp)
                    ];
                } ,
                VBox
                {   translateX: bind l.translateX;
                    translateY: bind l.boundsInParent.maxY+10;
                    spacing: 4;
                    content:
                    [   createRow(row1,false,createKey) ,
                        createRow(row2,true,createKey) ,
                        createRow(row3,false,createKey)
                    ];
                    effect: DropShadow
                    {    offsetX: 0;  offsetY: 5;
                    };
                }
            ];
        };
        fill: LinearGradient                                               C
        {   proportional: true;  endX: 0;  endY: 1;
            stops:
            [   Stop { offset: 0.0;  color:  Color.BURLYWOOD; } ,
                Stop { offset: 0.05;  color:  Color.BEIGE; } ,
                Stop { offset: 0.2;  color:  Color.SANDYBROWN; } ,
                Stop { offset: 0.9;  color:  Color.SADDLEBROWN; }
            ];
        }
        width: innerWidth;  height: innerHeight;
    }
```

```
        title: "Enigma";
        resizable: false;
        onClose: function() { FX.exit(); }
    }
    // Part three below
```
A Our new Paper class
B Rotors in horizontal layout
C Gradient backdrop to window

Listing 8.11 is the meat of the new changes. I've preserved the full listing this time, without any snips, so to better demonstrate how the changes integrate into what's already there. At the top of the scene graph we add our new `Paper` class, sized to be 50 pixels smaller than the window width. Directly below that we create a horizontal row of `Rotor` objects–recall the new `Rotor` is now a `CustomNode`, and can be used directly in the scene graph. The lamps have now been pushed down to be 5 pixels below the rotors.

At the very bottom of the `Scene` we install a `LinearGradient` fill, to create a pleasing shaded background for the window contents.

Just one more change to this class left, and that's in the on screen keyboard handler, as shown in listing 8.12.

Listing 8.12 Enigma.fx (version 2, part3 – changes only)

```
    // Part two above

    // [Snipped createRow(), createKey(),
    //   createLamp(): see previous version]

    function handleKeyPress(l:Integer,down:Boolean) : Void
    {   def res = encodePosition(l);
        (lamps.get(res) as Lamp).lit = down;
        if(not down)
        {   var b:Boolean;
            b=rotors[2].nextPosition();
            if(b) { b=rotors[1].nextPosition(); }
            if(b) { rotors[0].nextPosition(); }
        }
        else                                                          D
        {   paper.add(Rotor.posToChr(res));                           D
        }                                                             D
    }

    // [Snipped encodePosition(): see previous version]
```
D Key down? Add to paper

At last we have the final part of the Enigma class, and once again the unchanged parts have been snipped. Check out listing 8.7 if you want to refresh your memory. Only one change to mention, but it's an important one: the `handleKeyPress()` now has some plumbing to feed its key into the new `paper` node. This is what makes the encoded letter appear on the printout when a key is clicked.

8.3.4 Running version 2

Running the Enigma class gives us the display shown in figure 8.10 below. Our keys and lamps have been joined by a printout/teletype affair at the head of the window, and three rotors just below.

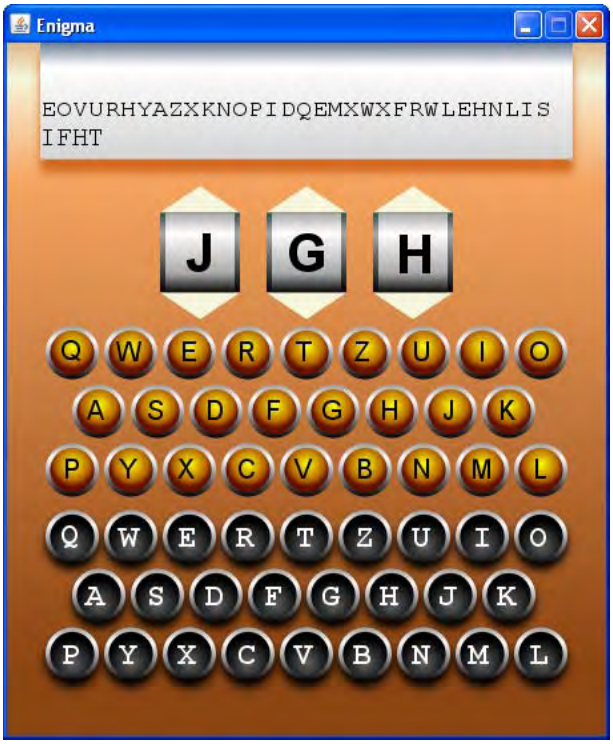


Figure 8.10 Our Enigma Machine in action, ready to keep all our most intimate secrets safe from prying eyes (providing they don't have access to any computing hardware made after 1940)

Clicking the arrows surrounding the rotors causes them to change position, while clicking the paper display causes it to jump up a line. As we stab away at the keys, our input is encoded through the rotors, flashed up on the lamps and appended to the printout.

Is our Enigma Machine finished, then? Well, for now, yes; but there's plenty of room for new features. The Enigma we developed is actually a simplified version of the original, with two important missing elements. The first is a plug board, twenty-six plug sockets joined by thirteen cables, the plug board could be thought of as another reflector, with connections which could be easily re-wired. Also, we have no way to change the order of the rotors. In the original machine the rotors were removable, and could be interchanged to further hinder

code-breaking. That said, unless anyone reading this book is planning on invading a small country, the simplified emulator we've developed should be more than enough.

In the final part of this chapter we'll turn our new application into an applet, so we can allow the world and his dog to start sending us secret messages.

8.4 From application to applet

One of the banes of deploying Java code in the *bad old days* was the myriad of options and mis-configurations one might encounter. Traditionally Java relied on the end user (the customer) to keep their installed Java implementation up to date, and naturally this lead to a vast range of runtime versions being encountered *in the wild*. The larger download size of the Java runtime, compared to the Adobe Flash Player, also put many web developers off. Java fans might note this is an apples and oranges comparison; the JRE is more akin to Microsoft's .NET Framework, and .NET is many times larger than the JRE. Yet still the perception of Java as large and slow persisted.

Starting with Java 6 Update 10, at the end of 2008, a huge amount of effort was put behind trying to smarten up the whole Java deployment experience, for both the end user and the developer. In the final part of this chapter we'll be exploring some of these new features, through JavaFX. Although none of them are specifically JavaFX changes, they go hand-in-glove with the effort to brighten up Java's prospects on the desktop, something which JavaFX *is* a key part of.

8.4.1 The Enigma class: from application to applet

Having successfully created an Enigma application, we need to port it over to be an applet. If you've ever written any Java applets this might send a chill down your spine. Although Java applications and applets have a lot in common, translating one to the other still demands a modest amount of reworking. But this is JavaFX, not Java, and our expectations are different. JavaFX was, after all, intended to ease the process of moving code between different types of user-facing environments.

Listing 8.13 shows the extent of the changes. Astute readers (those who actually read code comments) will note that once again the unchanged parts of the listing have been omitted, showing only the new material and its context.

Listing 8.13 Enigma.fx (version 3 – changes only)

```
package jfxia.chapter8;

import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.effect.DropShadow;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Stop;
```

```
import javafx.scene.shape.Rectangle;
import javafx.stage.AppletStageExtension;                                A
import javafx.stage.Stage;

import java.lang.System;
import java.util.ArrayList;

// [Snipped defintions for rotors, reflector, row1, row2, row3,
//   lamps, innerWidth, innerHeight and paper: see previous version]

Stage
{   // [Snipped Scene, title, resizable and
    //   onClose: see previous version]

    extensions:
    [   AppletStageExtension                                             B
        {   shouldDragStart: function(e: MouseEvent): Boolean            C
            {   return e.shiftDown and e.primaryButtonDown;              C
            }                                                            C
            useDefaultClose: true;                                       D
        }
    ];
}

// [Snipped createRow(), createKey(), createLamp(),
//   handleKeyPress() and encodePosition(): seee first version]
```

A **Import applet extension**
B **Plug exension into Stage**
C **Condition to begin drag action**
D **Closing X button**

You're eyes do not deceive you: the entire extent of the modifications amount to nothing more than eight lines of new code an an extra class import. The modifications center around an enhancement to the `Stage` object. To accommodate specific needs of individual platforms and environments, JavaFX supports an extension mechanism, using subclasses of `javafx.stage.StageExtension`. The specific extension we're concerned with is the `AppletStageExtension`, which adds functionality for using the JavaFX program from inside a web page.

As you can see from the code, `Stage` has an `extensions` property which accepts a sequence of `StageExtension` objects. We've used an instance of `AppletStageExtension`, which does two things. Firstly, it specifies when a mouse event should be considered the start of a drag action to rip the applet out of the web page and onto the desktop. Secondly, it adds a close box to the applet when on the desktop.

But simply changing our application isn't enough, for effective deployment on the web we need to package it into a JAR file. And that's just what we'll do, next.

8.4.2 The JavaFX Packager utility

The JavaFX SDK comes with a neat little utility to help deploy our JavaFX programs. It can be used to package up our applications for all manner of different environments. For our project we want to use it to target the desktop platform, and make the resulting program capable of running as an applet.

Netbeans users, take note

As always, I'm going to show you how the packaging process works under the hood, sans IDE. If you've using NetBeans, however, you may want to look over the following step-by-step tutorial for how to access the same functionality inside your IDE:

`http://javafx.com/docs/tutorials/deployment/`

The `javafxpackager` utility can be called from the command line, and has a multitude of options. The table 8.1, below, is a list of what's available.

Table 8.1 JavaFXPackager options

Option(s)	Function
`-src` or `-sourcepath`	The location of your Java source code (mandatory).
`-cp` or `-classpath` or `-librarypath`	The location of dependencies, extra JAR files your code relies upon.
`-res` or `-resourcepath`	The location of resource, extra data and images files your code relies upon.
`-d` or `-destination`	The directory to write the output.
`-workDir`	The temporary directory to use when building the output.
`-v` or `-verbose`	Print useful (debugging) information during the packaging process.
`-p` or `-profile`	Which environment to target. Either `DESKOP` or `MOBILE` (desktop includes applets).
`-appName` and `-appVendor` and `-appVersion`	Meta information about the application: its name, creator's name, and version.
`-appClass`	The startup class name, including package prefix (mandatory).
`-appWidth` and `-appHeight`	Application width and height (particularly useful for applets).

Option(s)	Function
`-appCodebase`	Codebase of where the application is hosted, if available on the web. This is the base address of where the JNLP, JAR and other files are located.
`-draggable`	Make applet draggable.
`-sign` and `-keystore` and `-keystorePassword` and `-keyalias` and `-keyaliasPassword`	Used to sign an applet to grant it extra permissions.
`-pack200`	Use Java-specific compression (usually much tighter than plain JAR).
`-help` and `-version`	Useful information about utility.

8.4.3 Packaging up the applet

Before we can run the packager for ourselves we need to make one change. In previous versions of this project we simply copied the FXZ files into the package directory created by our compiler. For the `javafxpackager` to work, however, we need to put them in their own resource directory.

1. Next to the "`src`" directory which holds the project source code, create a new directory called "`res`".
2. Inside the new `res` directory create one called "`jfxia`", and then one inside that called "`chapter8`". You should end up with a directory structure of `res/jfxia/chapter8` (or `res\jfxia\chapter8` if you prefer DOS backslashes) inside the project directory.
3. Copy the two FXZ files we're using for our project into the `chapter8` directory.

Now we're ready to run the packager. Open a new command console (such as an MS-DOS console on Windows) and change into the project directory, such that the `src` and `res` directories live off your current directory. Then type:

```
javafxpackager -src .\src -res .\res
  -appClass "jfxia.chapter8.Enigma"
  -appWidth 450 -appHeight 520
  -draggable -pack200 -sign
```

Alternatively, if you're running a Unix flavored machine, try this:

```
javafxpackager -src ./src -res ./res
  -appClass "jfxia.chapter8.Enigma"
```

```
-appWidth 450 -appHeight 520
-draggable -pack200 -sign
```

I've broken the command over several lines; you might want to turn it into a script (or batch file) if you plan on running the packager from outside an IDE often. Let's look at the options, chunk by chunk:

- `-src` is the location of our source code files. The packager attempts to build the source code, so it needs to know where it lives.
- `-res` is the location of any resource, which in our case is the `res` directory housing copies of our FXZ files.
- `-appClass` is the startup class for our application.
- `-appWidth` and `-appHeight` are the size of the applet.
- `-draggable` switches on the ability to drag the applet out of the browser (which we've catered for in our code).
- `-pack200` uses super tight compression for the resulting application archive.
- `-sign` signs the output so we can ask to be granted extra permissions when it runs on the end users machine. Even though our applet doesn't do anything dangerous, we may trigger a security problem when running it directly from the computer's file system (using a `file:` URL location). As we don't provide any keystore details, the packager will create a short term self-signed certificate for us, which will be fine for testing purposes.
- `-cp` (classpath) informs the packager about any extra JAR dependencies, so they can be included in the build process, and copied into the distribution output. We don't need any extra JARs, so I've not used this option (although if you're building the code under JavaFX 1.0 you'll need to reference the Production Suite's `javafx-fxd-1.0.jar` file).

If all went well, you should end up with a new directory, called "`dist`". This name is the default when we don't specify a `-destination` option. Inside, we have all the files we need to run the Enigma applet.

- `Enigma.jar` and `Enigma.jar.pack.gz`, which contain our project's classes and resources in plain JAR format, and Pack200 format.
- `Enigma.jnlp` and `Enigma_browser.jnlp`, which hold the Java Web Start details for our Enigma Machine in regular desktop, and applet, variants.
- `Enigma.html`, an HTML file we can use to launch our applet. We can copy the core markup from this file into any web page hosting the applet.
- `lib`, a directory with the JavaFX Production Suite's FXD library copied into it.

From the desktop, enter the `dist` directory and double click (or otherwise open) the HTML file. Your favorite web browser should start, and eventually the Enigma applet should

appear in all its glory. It may take some time initially, as the JavaFX libraries are not bundled in the distribution the package creates, but loaded over the web from the `javafx.com` site. This is to ensure maximum efficiency–why should every individual JFX application burden itself with local copies of the core API libraries, when they could be loaded (and maybe cached) from a single location across all JavaFX programs?

Figure 8.11 shows the applet in action.

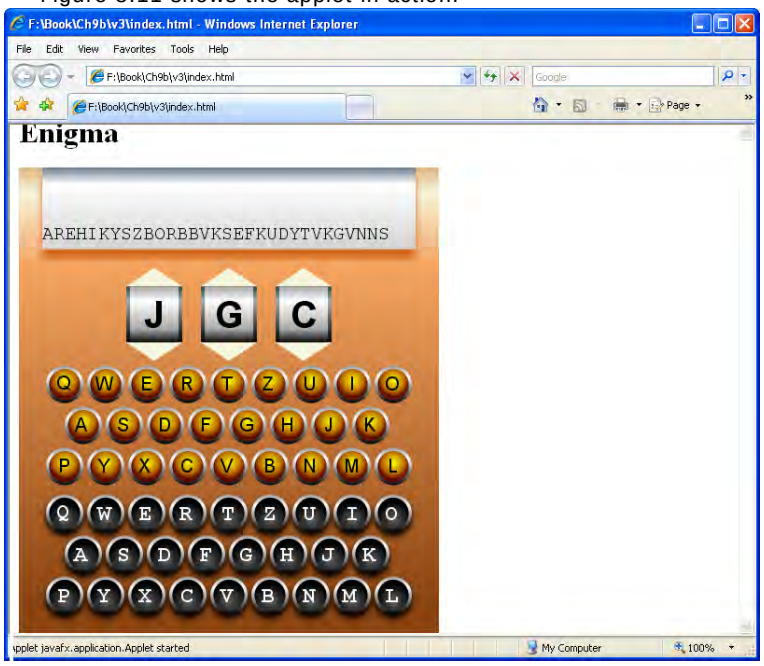


Figure 8.11 Our applet running inside Microsoft's Internet Explorer.

The Enigma machine looks just as it did when we ran it directly on the desktop, except now it's inside a browser window. Before we drag it back onto the desktop, let's have a quick look at the JNLP file, and make a small modification.

8.4.4 Dragging the applet onto the desktop

Dragging the application out of the browser gives us the opportunity to put an icon on the desktop. By default JavaFX uses a boring generic logo, but fortunately we can change that. To do so we need to create some icons files, and link them into the JNLP file.

If you download the source code from the book's web site you'll find, inside this project's directory, a couple of GIF files: `Enigma_32.gif` and `Enigma_64.gif` . We'll use these as our icons (unless you fancy creating your own?) Java Web Start allows us to specify icons of

different sizes, for use in different situations. For the record: the first of the supplied icons is 32x32 pixels in size, and the second is 64x64 pixels in size.

Copy these two images into the `dist` directory, and we've ready to change the JNLP. Take a look at listing 8.14.

Listing 8.14 Enigma_browser.jnlp

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+" codebase="file:/JavaFX_in_Action/dist/"
    href="Enigma_browser.jnlp">
    <information>
        <title>Enigma</title>
        <vendor>Simon</vendor>
        <homepage href=""/>
        <description>Enigma</description>
        <offline-allowed/>
        <shortcut>
            <desktop/>
        </shortcut>
        <icon href="./Enigma_32.gif" width="32" height="32" />
        <icon href="./Enigma_64.gif" width="64" height="64" />
    </information>
    <security>
        <all-permissions/>
    </security>
    <resources>
        <j2se version="1.5+"/>
        <property name="jnlp.packEnabled" value="true"/>
        <property name="jnlp.versionEnabled" value="true"/>
        <extension name="JavaFX Runtime"
            href="http://dl.javafx.com/1.1/javafx-rt.jnlp"/>
        <jar href="Enigma.jar" main="true"/>
    </resources>
    <applet-desc name="Enigma"
        main-class="com.sun.javafx.runtime.adapter.Applet"
        width="450" height="520">
        <param name="MainJavaFXScript" value="jfxia.chapter8.Enigma">
    </applet-desc>
</jnlp>
```

Above we have the `Enigma_browser.jnlp` file created by the packager for our application, complete with two extra lines (highlighted in bold) to hook up our icons. By studying the XML contents you can see how the details we passed into the packager were used to populate the JNLP configuration.

More JNLP information

Java Web Start supports all manner of options for controlling how Java applications behave as applets and on the desktop. Some of them are exposed by options on the JavaFX Packager, while others may require some post-packaging edits to the JNLP files. Unfortunately, an in depth discussion of JNLP is beyond the scope of this chapter. The

lengthy web address below (split over two lines) points to a page with details and examples of the various JNLP options.

```
http://java.sun.com/javase/6/docs/technotes/
  guides/javaws/developersguide/syntax.html
```

So much for the icons, now it's time to try dragging our applet out of the browser, and turning it back into a desktop application.

Double click the `Enigma.html` file inside `dist` to start it up again in the browser. Now hold down the SHIFT key on your computer, click and hold inside the applet with your mouse, then drag away from the browser. The applet should become unstuck from the browser page, and float over the desktop. Remember, we specified the criteria for the start of a drag operation in the `shouldDragStart` function of the `AppletStageExtension` we plugged into our application's `Stage`.

Once it's floating free, you can let the applet go, and it will remain on the desktop. Figure 8.12 shows how it might look. In place of the applet on the web page there should appear a default Java logo, showing where the applet once lived. In the top right corner of our floating applet there should be a tiny close button; a result of us specifying `useDefaultClose` in the extension.

Figure 8.12 Starting life in a web page, our Enigma emulator was then dragged onto the desktop to become an application (note the desktop icon), and finally re-launched from the desktop.

We now have two courses of action:

1. Clicking the applet's close button while the applet's web page is still open will return the applet back to its original home, inside the web page.
2. Clicking the web page's close button while the applet is outside the browser (on the desktop) will cause the browser window (or tab) to close and the applet to be installed onto our desktop, including a desktop icon.

Even though the applet's web page may have vanished, the applet continues to run without a hitch. It is now truly independent of the browser. This means getting the Enigma emulator onto the desktop is as simple as pulling it from the browser window, then closing the page. The applet will automatically transform into a desktop application, complete with startup icon (although in reality it remains a JavaWeb Start application, it just has a new home).

Removing the application should be as simple as following the regular un-install process for software on your operating system. For example, Windows users can use the “Add or

Remove Programs" program inside "Control Panel". Enigma will be listed along with other installed applications, with an button to remove it.

And that's it! At last we have a fully functional, JavaFX applet.

8.5 Summary

In this chapter we looked at writing a practical application, with a swish front end, focusing on a couple of important JavaFX skills. First of all we looked at how to take the hard toil of a friendly neighborhood graphic designer, bring it directly into our JavaFX project, and manipulate it as part of the JFX scene graph. Then we examined how to take our own hard toil, and magically transform it into an applet capable of being dragged from the browser and turned into a Web Start application. (Okay okay, it's not *actually* magic!)

As well as being a fun little project, I hope this chapter has demonstrated how easy it is to move JavaFX software from one environment to another, and how simple it is to move multimedia elements from artist to programmer. The addition of a `Stage` extension was all it took to add applet capabilities to our Enigma, and conversion into FXZ files was all it took to turn our vector images into fully programmable elements in our project.

In future there are plans to bring JavaFX to phones and digital TV technologies (Blu-ray DVD is already heavily Java influenced). The ability to leap across environments in a single bound would be a welcome change to the current drudgery of moving applications between platforms. Meanwhile, as software *everywhere* is become more graphically rich, a healthier interaction between design and code allows artists and engineers to collaborate without compromising their skills. It's this sense of freedom, both in *how* we work and *where* our code can run, which will be central to JavaFX as it evolves in the future.

So much for the future! For now, I'll just set the rotors to an appropriate three letters (I'll let you guess what they might be) and leave you with the simple departing message "NIAA CHZ ZBPS DVD AWOBHC RKNAHI".

Not a typesetter note: the final message is encoded with the letters JFX, and reads "HAVE FUN WITH OUR ENIGMA APPLET"

9

Getting mobile

Previous projects in this book have introduced a lot of fresh material and concepts, covering the core of the JavaFX Script language and its associated JavaFX APIs. Although we haven't explored every nook and cranny of JavaFX, by now you should have experienced a representative enough sample to navigate the remainder of the API without getting lost or confused. So this chapter is a deliberate shift down a gear or two.

Over the coming pages we won't be discovering any new libraries, classes, or techniques. Instead this chapter's project will focus on a couple of goals: we'll start by reinforcing the skills we're acquired thus far, pushing the scene graph in directions we've previously not seen; then we'll end by moving our finished project onto the JavaFX Mobile platform.

This book is being written against JavaFX release 1.1 (February 2009), and regrettably at this time the mobile emulator is only bundled with the Windows JavaFX SDK. This is an omission the JFX team has committed themselves to address soon; hopefully the fruits of their labor will be available by the time this book reaches you.

Even considering the limitations of the mobile preview, there's still a lot of value in the project outlined below. The code has been written to run in both the desktop and the mobile environments, so non-Microsoft uses can work through the majority of the project while waiting for the emulator to arrive on their platform.

The scene graph code we'll soon encounter is by far our most complex (and imaginative) yet, so once I've rattled through an outline of the project, we'll spend a little time on the secrets of how it's put together. But, first of all, we need to know what the project is.

9.1 Amazing games: a retro 3D puzzle

It's incredible to think, when looking at the sophistication of consoles like the Playstation and Xbox, just how far we've come in the last few decades. I'm just about old enough to remember the thrill of the Atari 2600, and the novelty of actually controlling an image on the TV set. The games back then seemed every bit as exciting as the games today, although

having replayed some of my childhood favorites thanks to modern day emulators, I'm at a loss to explain why. Some games never lose their charm, however. I discovered this the hard way a few years back, after losing the best part of a day to a Rubik's Cube I discovered while clearing out some old junk.

In this chapter we're going to develop a classic 3D maze game, like the one's which sprang up on the 8 bit computers a quarter of a decade before JavaFX was even a twinkle in Chris Oliver's eye. Retro games are fun, and popular with phone owners; not only is there an undeniable nostalgia factor, but their clean and simple game play fits nicely with the short bursts of activity typical of mobile device usage.

Figure 9.1 shows our retro maze game in action. The maze is really a grid, with some of the cells as walls and others as passageways. Moving through the maze is done one cell at a time, either forward, back, left or right. It's also possible to turn 90 degrees clockwise or anticlockwise. There's no swish true-3D movement here, the display simply jumps one block at a time, in true Dungeon Master fashion.

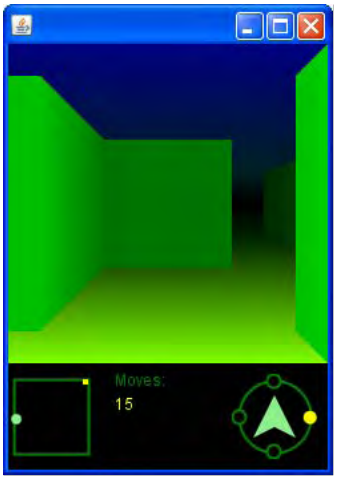


Figure 9.1 Get lost! This is our simple 3D maze game. Believe it or not the whole thing is constructed from the JavaFX scene graph, using basic shapes.

To aid the player we'll add a radar, similarly retro fashioned, giving an idea of where they are in relation to the boundaries of the maze. We'll also add a compass, spinning to show which way the view is facing. A game needs a score, so we'll track of the number of moves the player has made, the objective being to solve the maze in as few moves as possible.

The interface looks fairly simple, and that's because I want to spend most of the pages ahead concentrating on the 3D part of the game, rather than mundane control panel components. The 3D effect is really quite an unusual use of the scene graph, and it demands some careful forward-thinking and node management. So let's start by looking at the theory behind how it works.

9.1.1 Creating a faux 3D effect

One evening, many years ago, I wandered up the road where I lived at the time, and found myself transported back to Victorian London. A most surreal moment! It transpired a film crew had spent the morning shooting scenes for an episode of Sherlock Holmes, and the near by terrace housing had undergone a period makeover. If you've ever visited a Hollywood-style back lot you'll be familiar with how looks can deceive. Everything from the brick walls to the paved sidewalks are nothing more than lightweight fabrications, painted and *distressed* to make them look authentic.

This may surprise some readers, but the walls in our project do not use any mind bending 3D geometry. Just like the best Hollywood effects, they're all created with smoke and mirrors (not literally, obviously). The maze is nothing more than the illusion of 3D, creating the effect without any of the heavy lifting or number crunching. Now it's time to reveal its secrets.

Picture, if you will, five squares laid out side by side in a row. Now picture another row of squares, this time twice the size, overlaid and centered atop our original squares. And a third row, again twice as big (making them four times the size of the original row), and a fourth row (eight times the original size), all overlaid and centered onto the scene. If we joined the points from all these boxes, we might get a geometry like the one in figure 9.2.

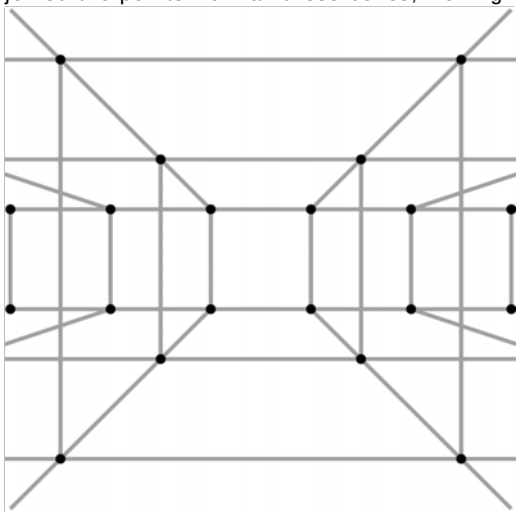
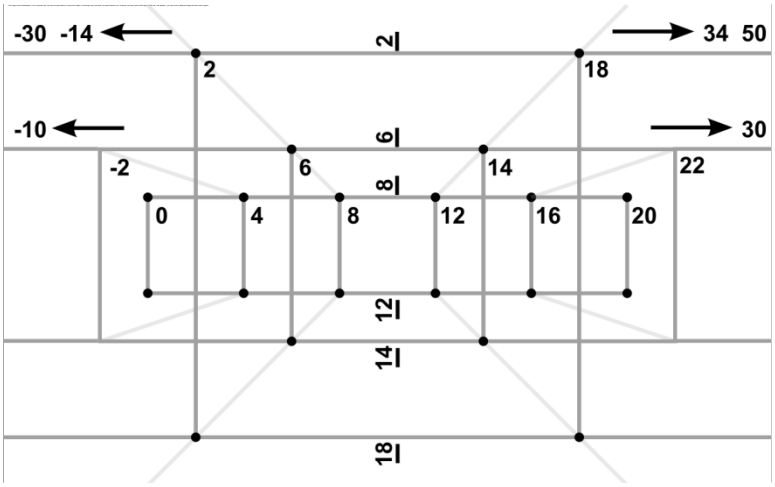


Figure 9.2 The 3D in our maze is all fake. The grid defines the maze geometry without using any complex mathematics.

Figure 9.2 has been clipped so that some of the squares are incomplete, or missing entirely. We can see the smallest row of squares, five in all, across the middle of the figure. The next row has three larger squares, the extremes of which are partially clipped. There are three even larger squares, so big only one fits fully within the clipped area, with just a



fragment of its companions showing. And the final row of sqares is so large they all fall entirely outside the clipping area, but we can see evidence for them in the diagonal lines leading off at the far corners of the figure.

This collection of sqares is all we need to construct the illusion of three dimensions.

9.1.2 Using 2D to create 3D

So, the 3D effect in our maze is entirely constructed from a plain and simple 2D grid, but how does that actually work? Figure 9.3 shows how the geometry of our 3D display is mapped to the 20x20 grid we're using to represent it.

Figure 9.3 The geometry of our maze. Using a flat 20x20 grid as the viewport, the regular numbers describe x co-ordinates, and the rotated numbers (underlined) describe y co-ordinates.

By using rows of squares, and connecting their points, we can build a faux 3D view. The visible area exposed in figure 9.2 fits within a grid, 20x20 points. Using a grid of that size we can express the coordinates of every square using integer values only. Figure 9.3, above, shows how those points are distributed inside (and outside) the 20x20 grid. Remember: even though the diagram *looks* 3D, the beauty is we're still dealing with 2D x/y points.

- Our five smallest squares (let's say they're the *furthest away*) are each 4 points wide, with the first one at (0,8)-(4,12), the second at (4,8)-(8,12), and so on.
- The next row of three squares are 8 points in size: (-2,6)-(6,16), then (6,6)-(14,14), and finally (14,8)-(22,14). The first and the last square falls slightly outside the (0,0)-(20,20) viewport.
- The next row of three is 16 points in size: (-14,2)-(2,18), then (2,2)-(18,18), etc. The first and last square fall predominantly outside the clipping viewport.
- The final row, not shown in figure 9.3, needs only one square which falls entirely outside the clipped area, at (-6,-6)-(26,26).

This simple grid of points is the key behind the custom node which creates the 3D maze display. Now we know the theory, we need to see how it works using code.

9.2 The Maze Game

The game we'll developer will be constructed from various custom nodes, centered around a *model* class for the maze and player status. The scene graph code is pretty simple, except for the 3D node, which is highly complex. The 3D custom node demonstrates just how far we can push JavaFX's *retained mode* graphics, with clever positioning and management of nodes, to create a display more typical of *immediate mode* (the model Swing uses).

9.2.1 The MazeDisplay class: 3D view from 2D points

We might was well start with the most complex part of the UI, indeed the most involved piece of UI code we've seen so far in this book. The `MazeDisplay` class is the primary custom node responsible for rendering the 3D view. As with previous source code listings, this one has been broken into stages. The first of these is listing 9.1.

Listing 9.1 MazeDisplay.fx (part 1)

```
package jfxia.chapter9;

import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Stop;
import javafx.scene.shape.Polygon;
import javafx.scene.shape.Rectangle;


package class MapDisplay extends CustomNode
{   def xPos:Integer[] =                                          A
    [   0,  4,  8, 12, 16, 20 ,                                   A
      -10, -2,  6, 14, 22, 30 ,                                   A
      -30,-14,  2, 18, 34, 50 ,                                   A
      -70,-38, -6, 26, 58, 90                                     A
    ];                                                            A
    def yPos:Integer[] =                                          B
    [   8,  6,  2, -6 ,                                           B
       12, 14, 18, 26                                             B
    ];                                                            B

    public-init var map:Map;                                      C

    var wallVisible:Boolean[];                                    D
    def scale:Integer = 12;                                       E
// Part two below
```

A Horizontal points on faux 3D
B Vertical points on faux 3D
C Map data
D Manipulates scene graph
E Scale points on screen

Above we have the opening to our maze view class, and already we're seeing some pretty crucial code for making the 3D effect work.

The opening tables, xPos and yPos, represent the points on the 20x20 grid we saw in figure 9.3. Each line in the xPos table represents the horizontal coordinate for five squares. Even though *nearer* (larger) rows only require three squares or one square, we use five consistently to balance the table and make it easier to access. Each line contains six entries because we need not just the left side coordinate, but the right side too; the final entry defines the right hand side of the final square (and the left side of the sixth box, of course, which is surplus to our requirements). The first line in the table represents the row of squares furthest away (smallest), with each successive line describing nearer (larger) rows.

The yPos table documents the vertical coordinate for each row. The first line describes the y positions for the tops of the furthest (smallest) row, through to the nearest (largest) row. The next line does the same for the bottoms.

We'll be using these two tables when we build our scene graph, in a short while. Meanwhile, let's consider the remainder of the variables. The map variable is where we get our maze data from; it's an instance of the Map class, which we'll see later. The wallVisible boolean sequence is used to manipulate the scene graph once it has been created. Finally, the scale decides how many pixels each point in our 20x20 grid is worth. Setting scale to 12 results in a maze display of 240x240 pixels, ideal for our mobile environment.

The next step is to create the application's scene graph. Listing 9.2 shows how it fits together.

Listing 9.2 MazeDisplay.fx (part 2)

```
// Part one above
    override function create() : Node
    {   def n:Node = Group
        {   content:
            [   Rectangle
                {   width: 20*scale;
                    height: 20*scale;
                    fill: LinearGradient                                    F
                    {   proportional: true;
                        endX: 0.0;  endY: 1.0;
                        stops:
                        [   Stop { offset: 0.0;
                                color: Color.color(0,0,.5); },
                            Stop { offset: 0.40;
                                color: Color.color(0,.125,.125); },
                            Stop { offset: 0.50;
                                color: Color.color(0,0,0); },
                            Stop { offset: 0.60;
                                color: Color.color(0,.125,0); },
                            Stop { offset: 1.0;
                                color: Color.color(.5,1,0); }
                        ];
                    };
                } ,
                _wallFront(0,4,0 , 0.15),                                   G
                _wallSides(1,4,0 , 0.15,0.45),                              H
                _wallFront(1,3,1 , 0.45),                                   H
```

```
                _wallSides(2,3,1 , 0.45,0.75),                          I
                _wallFront(1,3,2 , 0.75) ,                              I
                _wallSides(2,3,2 , 0.75,1.00)                           J
            ]
            clip: Rectangle
            {   width: 20*scale;
                height: 20*scale;
            };
            cache: true;                                                K
        };
        update();                                                       L

        n;
    }
// Part three below
```

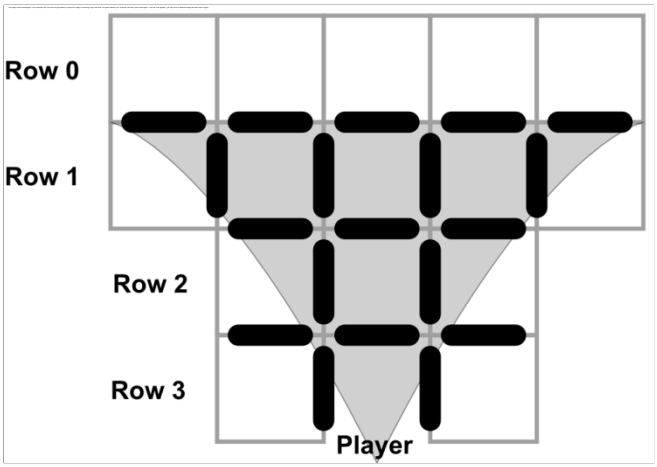
F Sky and floor gradient
G Smallest row, fronts
H Medium row, sides/fronts
I Large row, sides/fronts
J X-Large sides
K Cache output
L Populate scene with map data

The `create()` function should, by now, be immediately recognizable as the code which creates our scene graph. It begins with a simple background `Rectangle`, using a `LinearGradient` to paint the sky and floor, gradually fading off to darkness as they approach the horizon. To populate the colors in the gradient we're using a script level (static) function of `Color`, which returns a hue based upon red, green and blue values expressed as decimals between zero and one.

After the background a series of functions calls populate the graph with front facing and side facing walls. These are just convenience functions which help keep `create()` looking nice and clean (I tend to prefix the name of such *refactored* functions with an underscore, although that's just personal style). We'll look at the mechanics of how the shapes are added to the scene in a moment, but for now let's just focus on the calls themselves. Here they are again:

```
_wallFront(0,4,0 , 0.15),
_wallSides(1,4,0 , 0.15,0.45),
_wallFront(1,3,1 , 0.45),
_wallSides(2,3,1 , 0.45,0.75),
_wallFront(1,3,2 , 0.75) ,
```

The first call is to `_wallFront()`. It adds in the back row of five front facing walls, which you can see in figure 9.4 below. The three integer parameters all relate to the tables be saw in listing 9.1. The first two parameters are delimiters, from 0 to 4 (five walls), and the third parameter determines which parts of the `xPos` and `yPos` tables to use for coordinates. In this case we're using line 0 (the first six entries) in xPos, and entry 0 in the top/bottom lines from `yPos`. In plain English this means we'll be drawing boxes 0 to 4 using horizontal coordinates 0, 4, 8, 12, 16 and 20 from `xPos`, and vertical coordinates 8 and 12



from yPos. The final, fractional, parameter determines how dark to make each wall. As these walls are furthest away, they are set to a very low brightness.

The next line is a call to _wallSides(). It adds in the four side facing walls, two left of center, two right of center. The first three parameters do pretty much the same thing, but there are two brightness parameters. This is because the 'perspective' walls (the ones running *into* the screen) have a different brightness from their furthest to their nearest edge. The first parameter is for the furthest, the second is for the nearest.

```
_wallSides(2,3,2 , 0.75,1.00)
```

Figure 9.4 A plan view of the scene graph pieces which have to be added, in order, from back (row 0) to front (row 3). The shaded area represents the players field of view.

The remaining function calls add in the other front and side walls, working from the back to the front of the 3D scene. Next, we see the actual function code (listing 9.3).

Listing 9.3 MazeDisplay.fx (part 3)

```
// Part two above
    function _wallFront(x1:Integer,x2:Integer , r:Integer,
    op:Number ) : Node[]
    {   for(x in [x1..x2])                                          M
        {   insert false into wallVisible;                          N
            def pos:Integer = sizeof wallVisible -1;                N

            Polygon                                                 O
            {   points:
                [   xPos[r*6+x+0]*scale , yPos[0+r]*scale , // UL
                    xPos[r*6+x+1]*scale , yPos[0+r]*scale , // UR
                    xPos[r*6+x+1]*scale , yPos[4+r]*scale , // LR
                    xPos[r*6+x+0]*scale , yPos[4+r]*scale   // LL
                ];
                fill: Color.color(0,op,0);
                visible: bind wallVisible[pos];                     P
            };
        }
    }

    function _wallSides(x1:Integer,x2:Integer , r:Integer,
    opBack:Number,opFore:Number) : Node[]
    {   var half:Integer = x1 + ((x2-x1)/2).intValue();
        for(x in [x1..x2])                                          Q
        {   def rL:Integer = if(x>half) r else r+1;                 R
            def rR:Integer = if(x>half) r+1 else r;                 R
```

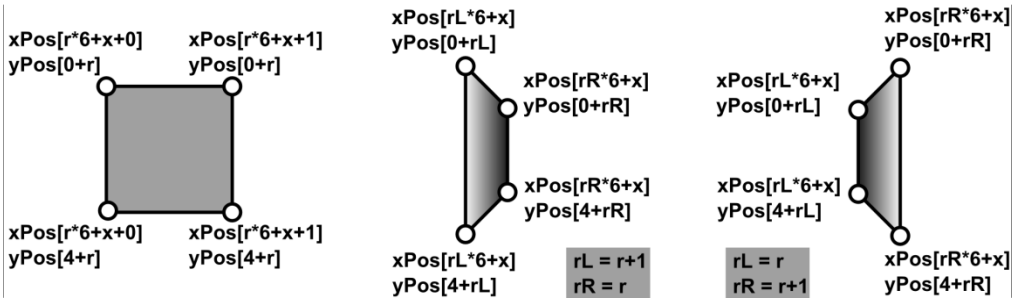
```
            def opL:Number = if(x>half) opBack else opFore;              S
            def opR:Number = if(x>half) opFore else opBack;              S

            insert false into wallVisible;                              T
            def pos:Integer = sizeof wallVisible -1;                    T

            Polygon                                                     U
            {   points:
                [   xPos[rL*6+x]*scale , yPos[0+rL]*scale , // UL
                    xPos[rR*6+x]*scale , yPos[0+rR]*scale , // UR
                    xPos[rR*6+x]*scale , yPos[4+rR]*scale , // LR
                    xPos[rL*6+x]*scale , yPos[4+rL]*scale   // LL
                ];
                fill: LinearGradient
                {   proportional: true;
                    endX:1;  endY:0;
                    stops:
                    [   Stop                                            V
                        {   offset:0;                                   V
                            color:Color.color(0,opL,0);                 V
                        } ,                                             V
                        Stop                                            W
                        {   offset:1;                                   W
                            color:Color.color(0,opR,0);                 W
                        }                                               W
                    ];
                };
                visible: bind wallVisible[pos];                         X
            };
        }
    }
// Part four below
```

M For each wall
N Add on/off switch
O Front wall polygon
P Bind to on/off switch
Q For each wall
R Near / far edge?
S Near / far brightness?
T Add on/off switch
U Side wall polygon
V Left edge color
W Right edge color
X Bind to on/off switch

Listing 9.3 shows the two functions responsible for adding the walls into our scene graph. The first function, `_wallFront()`, adds the front facing walls. It loops inclusively between two delimiters, parameters `x1` and `x2`, adding `Polygon` shapes. The `r` parameter determines which row in the view we're drawing, which in turn determines the parts of the `xPos` and `yPos` tables we should use. For example, when `r=0` the table data for the furthest (smallest) row of walls is used, when `r=3` the nearest (largest) row is used.

The polygon forms a square using the coordinate pairs upper-left, upper-right, lower-right and lower-left, in that order. We could have used a `Rectangle`, but using a `Polygon` makes the code more consistent with its companion `_wallSides()` function (see later).

Because the xPos and yPos tables are, in reality, linear sequences, we need to do a little



bit of math to find the correct value. Each row of horizontal coordinates in xPos has six points, from left to right across the view. There are two sets of vertical coordinates (upper y and lower y) in yPos, each with four points (rows 0 to 3). For horizontal coordinates we multiply r up to the right row, then add on the wall index to find the left hand side, or its next door neighbor for the right hand side. The vertical coordinates, which define the top and bottom of the shape, are as simple as reading the r'th value from the first and second line of yPos.

Figure 9.5 shows how the x and y coordinates are looked up inside the two tables. Remember, once we've used the tables to find the point in our 20x20 grid, we need to apply scale to multiply it up to screen pixel coordinates.

Figure 9.5 By plotting the points on our polygon, using the xPos and yPos tables for reference, we can created the illusion of perspective.

Just before we create the polygon, we add a new boolean to wallVisible, and in the polygon itself we bind visibility to its index. This allows us to switch the wall on or off (visible or invisible) later in the code. This switch is key to updating the 3D view, as demonstrated in the final part of the code, below.

The _wallSides() function is a more complex variation of the function we've just studied. This time, however, one edge of the polygon is on one row, and the other edge is on another. To decide which edge is where we need to know whether we're drawing a given shape on the left hand side of the scene, or the right, to ensure we get the perspective correct. The half variable is used to that purpose, and instead of one r value we have two: one for the left and one for the right. We also have two color values for the edges of the polygon, to ensure it gets darker as it goes *deeper* into the display.

Once again we add a boolean to wallVisible, and bind the polygon's visibility to it. But what do we do with all these booleans? The answer is up next in listing 9.4.

Listing 9.4 MazeDisplay.fx (part 4)

```
// Part three above
    package function update() : Void
    {   def walls:Integer[] =
```

```
        [   -2,2,-3 ,                                                   Y
            -2,-1,-2 , 1,2,-2 ,                                         Z
            -1,1,-2 ,                                                  AA
            -1,-1,-1 , 1,1,-1 ,                                        BB
            -1,1,-1 ,                                                  CC
            -1,-1,0 , 1,1,0                                            DD
        ];

        var idx:Integer = 0;
        var pos:Integer = 0;
        while(idx<sizeof walls)
        {   var yOff:Integer = walls[idx+2];                           EE
            for(xOff in [walls[idx]..walls[idx+1]])                    EE
            {   var rot:Integer[] = map.rotateToView(xOff,yOff);
                wallVisible[pos] = not map.isEmpty                     FF
                    (map.x+rot[0] , map.y+rot[1]);                     FF
                pos++;
            }
            idx+=3;                                                    GG
        }
    }
}
```

Y Row 0, fronts
Z Row 1, sides
AA Row 1, fronts
BB Row 2, sides
CC Row 2, fronts
DD Row 3, sides
EE (x1,y) to (x2,y)
FF Change visibility
GG Next "x1, x2, y" triple

This is our final chunk of the mammoth 3D code. Having put all our polygons in place, we need to update the 3D view by switching their visibility on or off, depending upon which cells of the map inside our viewport are walls and which are empty (passageways). We do this each time the player moves, using the `update()` function.

A quick reminder: we added rows of polygons into our scene graph, in sequence, from furthest to nearest. At the same time we added booleans to `wallVisible` in the same order, tying them to the corresponding polygon's visibility. These booleans are the means by which we will now manipulate each wall polygon. As we move around the maze different parts of the map fall inside our field of view (the viewport). The wall nodes never get moved or deleted, when a give cell of the map has no wall block present, the corresponding polygons are simply made invisible (they're still there in the graph, they just don't get drawn).

This nifty piece of code looks at map positions, relative to the player's position and orientation (facing north, south, east or west), and sets the visibility boolean on the corresponding node of the scene graph. Figure 9.6 shows how the relative coordinates relate to the player's position at (0,0).

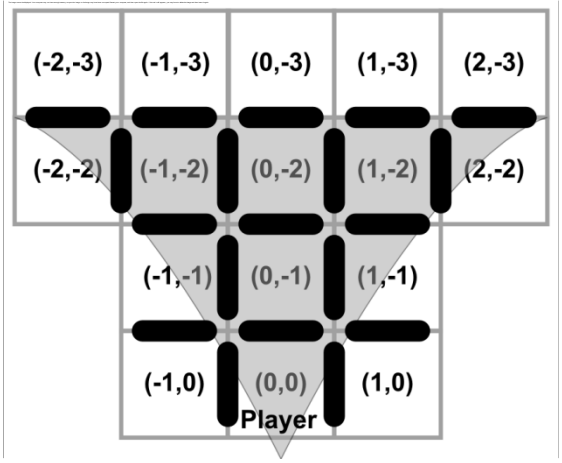


Figure 9.6 Having creating our scene graph walls, we need to be able to switch them off and on depending upon which cells in the map are wall blocks, relative to the player's location.

We need to work through each polygon node in the scene graph, relate it to an x/y coordinate in the map, and set the corresponding `wallVisible` boolean to turn the polygon on or off. The `walls` table at the start of the code holds sets of x1, x2, and y triples which control this process. Let's remind ourselves of the table, with one eye on figure 9.6, so we can see what that means:

```
def walls:Integer[] =
[   -2,2,-3 ,
    -2,-1,-2 , 1,2,-2 ,
    // Snipped
]
```

The first nodes in the scene graph are the back walls, which run from coordinates (-2,-3) to (2,-3) relative to our player's position; where (0,0) is the cell the player is currently standing on. This is why our first triple is -2,2,-3 (x=-2 to x=2, with y=-3).

Next we have four side walls, controlled by cells (-2,-2) and (-1,-2) on the left hand side of the view, then (1,-2) and (2,-2) on the right. And you can see the second line of `walls` has the corresponding triples. (We don't bother with the central cell, (0,-2), because that cell's side walls are entirely obscured by its front facing polygon.)

The remaining lines in `walls` deals with the rest of the scene graph, row by row. All of these values are fed into a loop, translating the table into on/off settings for each wall polygon.

```
var idx:Integer = 0;
var pos:Integer = 0;
while(idx<sizeof walls)
{   var yOff:Integer = walls[idx+2];
    for(xOff in [walls[idx]..walls[idx+1]])
    {   var rot:Integer[] = map.rotateToView(xOff,yOff);
        wallVisible[pos] = not map.isEmpty
            (map.x+rot[0] , map.y+rot[1]);
        pos++;
    }
    idx+=3;
}
```

The `while` loop works its way through the `walls` table, three entries at a time; the variable `idx` storing the current offset into `walls`. Remember, the first value in the triple is the start x coordinate, the second is the end x coordinate, and the third is the y coordinate. The nested `for` loop then works through these coordinates, rotating them to match the player's direction, and adding them into the current player x/y position to get the absolute cell in the map to query. We then query the cell to see if it is empty or not.

Once this code has run, each of the polygons in the scene graph will be visible or invisible, depending upon its corresponding entry in the map.

Voila, our 3D display is alive!

Whew! I did warn you this was the most complex piece of scene graph code in the book, and I'm sure it didn't disappoint. You may have to scan through the details a couple of times just to ensure they sink in. The purpose behind this example is to show that retained mode graphics (scene graph based, in other words) don't have to be simple shapes. They are capable of complexity similar to immediate mode graphics. All it takes is a little planning, and imagination.

You'll be glad to hear the remaining custom nodes are trivial by comparison.

9.2.2 The Map class: where are we?

Now we've seen the `MapDisplay` class, its time to introduce the *model* class which provides its data. Listing 9.5 provides the code.

Listing 9.5 Map.fx (part 1)

```
package jfxia.chapter9;

package class Map
{   def wallMap:Integer[] =                                                A
    [   1,1,1,1,1,1,1,1,1,1,1,1,1,1,1 ,                                    A
        1,0,0,1,0,1,0,0,0,0,0,0,0,1,0,1 ,                                  A
        1,1,0,1,0,1,1,1,0,1,1,0,1,1,0,1 ,                                  A
        1,0,0,1,0,1,0,1,0,0,0,0,0,1,0,1 ,                                  A
        1,0,1,1,0,1,0,1,0,1,1,1,0,0,0,1 ,                                  A
        1,0,0,1,0,0,0,1,0,1,0,1,1,1,1,1 ,                                  A
        1,0,0,1,1,1,1,1,0,1,0,0,0,0,0,1 ,                                  A
        1,0,1,1,0,0,0,1,0,1,1,1,0,1,0,1 ,                                  A
        1,0,0,1,1,0,1,1,0,0,0,1,0,1,1,1 ,                                  A
        1,1,0,0,0,0,0,1,0,1,1,1,0,1,0,1 ,                                  A
        1,0,0,1,1,0,1,1,0,0,1,0,0,0,0,1 ,                                  A
        1,1,0,0,1,0,0,1,1,0,1,0,1,0,0,1 ,                                  A
        1,0,0,1,1,1,0,0,0,0,0,1,1,0,1 ,                                    A
        1,0,0,0,1,0,0,1,1,0,1,1,1,0,0,1 ,                                  A
        1,1,0,0,0,1,0,1,0,0,0,0,0,0,0,1 ,                                  A
        1,1,1,1,1,1,1,1,1,1,1,1,1,1,1                                      A
    ];                                                                     A

    package def width:Integer = 16;                                        B
    package def height:Integer = 16;                                       B

    package def startX:Integer = 1;                                        C
    package def startY:Integer = 1;                                        C
```

```
    package def endX:Integer = 14;                                    C
    package def endY:Integer = 1;                                     C

    public-read var x:Integer;                                        D
    public-read var y:Integer;                                        D
    public-read var dir:Integer;                                      D
    public-read var success:Boolean =                                 E
        bind ((x==endX) and (y==endY));                               E
// Part two below
```

A Maze wall data. 1 is a wall, 0 is an empty space.
B Dimensions of map
C Start and end cells
D Current player position/direction
E Reached the end yet?

Above we have the first half of our map class. It begins with a huge table defining which parts of the map are walls, and which passageways. Immediately following that we have the dimensions of the map, 16x16 cells., followed by the cells which denote the start and end locations.

The `public-read` variables expose the current cell (x and y) the player is standing on, and which direction they are facing. North is 0, east is 1, south is 2 and west is 3. Finally we have a convenience boolean, true when the player's location is the same as the end location (and the maze has therefore been solved).

Let's move on the listing 9.6, which is the second part of the `Map` class.

Listing 9.6 Map.fx (part 2)

```
// Part one above
    init
    {   x = startX;
        y = startY;
    }

    package function isEmpty(x:Integer,y:Integer) : Boolean
    {   if(x<0 or y<0 or x>=width or y>=height) { return false; }
        var idx:Integer = y*width+x;
        return( wallMap[idx]==0 );
    }

    package function moveRel(rx:Integer,ry:Integer,rt:Integer) : Boolean
    {   if(rx!=0 or ry!=0)                                            F
        {   def rot:Integer[] = rotateToView(rx,ry);                  G
            if(isEmpty(x+rot[0],y+rot[1]))                            H
            {   x+=rot[0];  y+=rot[1];                                H
                return true;                                          H
            }                                                         H
            else
            {   return false;
            }
        }
        else if(rt<0)                                                 I
        {   dir=(dir+4-1) mod 4;                                      I
            return true;                                              I
        }                                                             I
```

```
            else if(rt>0)                                                    J
            {   dir=(dir+1) mod 4;                                           J
                return true;                                                 J
            }                                                                J
            else
            {   return false;
            }
        }

        package function rotateToView(x:Integer,y:Integer) : Integer[]
        {   [   if(dir==1) 0-y                                               K
                    else if(dir==2) 0-x                                      K
                    else if(dir==3) y                                        K
                    else x ,                                                 K
                if(dir==1) x                                                 L
                    else if(dir==2) 0-y                                      L
                    else if(dir==3) 0-x                                      L
                    else y                                                   L
            ];
        }
    }
```

F Moving x/y ?
G Rotate coordinates
H If possible, move to cell
I Turn left (anticlockwise)
J Turn right (clockwise)
K Calculate absolute x coordinate
L Calculate absolute y coordinate

In the conclusion of the `Map` class we have a set of useful functions for querying the map data, and updating the player's position.

- The `init` block assigns the player's current location from the map's start location.
- The `isEmpty()` function returns true if the cell at `x` and `y` has no wall. We saw it in action during the update of the 3D maze, determining whether nodes should be visible or invisible.
- The `moveRel()` function accepts three parameters, controlling `x` movement, `y` movement, and rotation. It returns true if the move was performed. The parameters are used to move the player relative to the current position **and orientation**, or rotate their view. If `rx` or `ry` are not zero, the relative positions are added to the current player location (the `rx` and `ry` coordinates are based on the direction the player is currently facing, so the `rotateToView()` function orientates them the same way as the map). If the `rt` parameter is not zero, it is used to rotate the player's current orientation.
- Finally, `rotateToView()` is the function we've seen used repeatedly whenever we needed to translate relative coordinates facing in the player's current orientation to relative coordinates orientated north. It returns a sequence of two values, `x` and `y`. For example, if I move forward one cell then the relative movement is `x=0, y=-1` (vertically one cell up, horizontally no cells). But if I'm

facing east at the time, this needs translating to x=1, y=0 (horizontally right one cell, vertically no cells) to make sense on the map data.

So that's our map class. There are three simple custom nodes we need to look at before we pull everything together into our main application. So let's deal with them quickly.

9.2.3 The Radar class: this is where we are

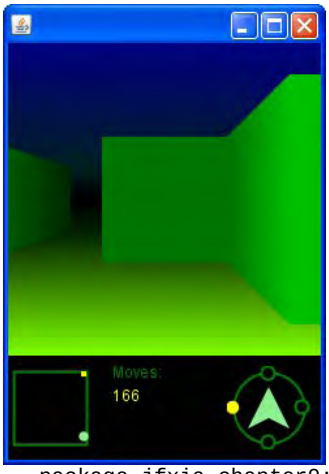
The `Radar` class is a simple class for displaying the position of the player within the bounds of the maze, as show in the bottom left corner of figure 9.7.

Figure 9.7 The maze game, complete with radar in the bottom left hand corner, and a compass in the bottom right.

The radar has a kind of 8 bit retro style, in keeping with the simple, unfussy, graphics of the maze.. It doesn't show the walls of the maze, that would make the game too easy, just a pulsing circle representing where the player is, and a yellow square for the goal.

Let's take a look at the code, courtesy of listing 9.7.

Listing 9.7 Radar.fx




```
package jfxia.chapter9;

import javafx.animation.transition.ScaleTransition;
import javafx.animation.Timeline;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Rectangle;
```

```
package class Radar extends CustomNode
{   def cellSize:Number = 4;
    def border:Number = 8;
    public-init var map:Map;

    override function create() : Node
    {   var c:Circle;
        var n:Group = Group
        {   def w = map.width*cellSize;
            def h = map.height*cellSize;

            translateX: border;
            translateY: border;
            content:
            [   Rectangle                                                    A
                {   width: w;  height: h;
                    fill: null;
                    stroke: Color.GREEN;
                    strokeWidth:2;
                } ,
                c = Circle                                                   B
                {   translateX: cellSize/2;
                    translateY: cellSize/2;
                    centerX: bind map.x * cellSize;
                    centerY: bind map.y * cellSize;
                    radius: cellSize;
                    fill: Color.LIGHTGREEN;
                } ,
                Rectangle                                                    C
                {   x: map.endX * cellSize;
                    y: map.endY * cellSize;
                    width: cellSize;
                    height: cellSize;
                    fill: Color.YELLOW;
                }
            ];
            clip: Rectangle
            {   width: w + border*2;
                height: h + border*2;
            }
        }

        ScaleTransition                                                      D
        {   node: c;                                                         D
            duration: 0.5s;                                                  D
            fromX: 0.2;  fromY: 0.2;                                         D
            toX: 1;  toY: 1;                                                 D
            autoReverse: true;                                               D
            repeatCount: Timeline.INDEFINITE;                                D
        }.play();                                                            D

        n;
    }
}
```

A Background rectangle

B Circle presenting player
C End marker
D Infinite scale in/out

The `Radar` class has a very simple scene graph combined with a `ScaleTransition`. The `cellSize` is the pixel dimension each maze cell will be scaled to on our display. You'll recall from the `Map` class, the maze is 16x16 cells in size. At a pixel size of 4, this means the radar will be 64x64 pixels. And talking of the `Map` class, the `map` variable is a local reference to the game's state and data. The `border` is the gap around the edge of the radar, bringing the total size to 80x80 pixels.

The scene graph manufactured in `create()` is very basic, but then it doesn't need to be fancy. Three shapes are contained within a `Group`, translated by the `border`.

The first shape, a background `Rectangle`, provides the boundary of the maze area, sized using the `map` dimensions against the `cellSize` constant. A `Circle` is used to show where the player currently is; its position is bound to the `cellSize` multiplied by the player's location in the `map`. Because a JavaFX `Circle` node's origin is its center point, we translate it by half a cell, to put the origin of the circle in the center of the cell. Finally, the end point in the map (the winning cell) is displayed using a yellow `Rectangle`.

Before we return the `Group` we create a `ScaleTransition`, continually growing and shrinking the circle from full size to just 20%. With the `autoReverse` flag set to true, the animation will play forwards then backwards. And with `repeateCount` set to `Timeline.INDEFINITE`, it will continue to run, forwards and back, forever. (Well, at least until the program exits!)

9.2.4 The Compass class: this is where we're facing

The `Compass` class is another simple, retro style, class. This one spins to point in the direction the player is facing.

Listing 9.8 is our compass code. It's a two part custom node, with a static part which does not move, and a mobile part which rotates to show the player's orientation. You can see what it looks like by glancing back to figure 9.7.

Listing 9.8 Compass.fx

```
package jfxia.chapter9;

import javafx.animation.transition.RotateTransition;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Polygon;
import javafx.scene.shape.Rectangle;

package class Compass extends CustomNode
{   def size:Number = 64;
    def blobRadius:Number = 5;

    public-init var map:Map;
```

```
var compassNode:Node;

override function create() : Node
{   def sz2:Number = size/2;
    def sz4:Number = size/4;

    compassNode = Group                                                    A
    {   content:
        [   Circle                                                         B
            {   centerX: sz2;
                centerY: blobRadius;
                radius: blobRadius;
                fill: Color.YELLOW;
            } ,
            _makeCircle(blobRadius , sz2) ,                                C
            _makeCircle(size-blobRadius-1 , sz2) ,                         C
            _makeCircle(sz2 , size-blobRadius-1)                           C
        ]
        rotate: map.dir * 90;                                              D
    };

    Group                                                                  E
    {   content:
        [   Circle                                                         F
            {   centerX: sz2;  centerY: sz2;                               F
                radius: sz2-blobRadius;                                    F
                stroke: Color.GREEN;                                       F
                strokeWidth: 2;                                            F
                fill: null;                                                F
            } ,                                                            F

            Polygon                                                        G
            {   points:                                                    G
                [   sz2 , sz4 ,                                            G
                    size-sz4 , size-sz4,                                   G
                    sz2 ,  sz2+sz4/2 ,                                     G
                    sz4 , size-sz4                                         G
                ];                                                         G
                fill: Color.LIGHTGREEN;                                    G
            } ,                                                            G
            compassNode                                                    H
        ]
        clip: Rectangle { width:size;  height:size; }
    }
}

package function update() : Void                                           I
{   RotateTransition
    {   node: compassNode;
        duration: 1s;
        toAngle: map.dir * 90;                                             J
    }.play();
}

function _makeCircle(x:Number,y:Number) : Circle                           K
{   Circle
```

```
            {   centerX: x;  centerY: y;
                radius: blobRadius;
                stroke: Color.GREEN;
                strokeWidth: 2;
            }
        }
    }
```

A Rotating group
B North circle (yellow)
C West, east and south circles (hollow)
D Initial rotation
E Static group
F Ring circle
G Central arrow
H Add in rotating group
I Change rotation
J Animate to new direction
K Convenience function: make a circle

The instance variable `size` is the diameter of the ring, while `blobRadius` helps size the circles around the ring. (Blob?! Well, can *you* think of a better name?)

Inside `create()`, the `compassNode` group is the rotating part of the graph. It is nothing more than four circles (blobs?) representing the points of the compass, with the northern position a solid yellow color. The other three are hollow, and identical enough to be created by a convenience function, `_makeCircle()`. The `compassNode` is plugged into a static part of the scene graph, which features a ring and an arrow polygon.

The `update()` function is called whenever the direction the player if facing changes. It kicks off a `RotateTransition` to spin the blob group, making the yellow blob face the player's direction. Because we only specify an end angle, and no starting angle, the transition (conveniently) starts from the current rotate position, whatever that may be.

9.2.5 The ScoreBoard class: are we there yet?

Only one more custom node to go, then we can write the application class itself. The scoreboard keeps track of the score, and displays a winning message; its code is in listing 9.9.

Listing 9.9 ScoreBoard.fx

```
package jfxia.chapter9;

import javafx.scene.CustomNode;
import javafx.scene.Node;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Text;
import javafx.scene.text.TextOrigin;


package class ScoreBoard extends CustomNode
{   public-init var score:Integer;
    package var success:Boolean = false;
```

```
    override function create() : Node
    {   VBox
        {   spacing: 5;
            content:
            [   Text                                                    A
                {   content: "Moves:";                                  A
                    textOrigin: TextOrigin.TOP;                         A
                    fill: Color.GREEN;                                  A
                } ,                                                     A
                Text                                                    B
                {   content: bind "{score}";                            B
                    textOrigin: TextOrigin.TOP;                         B
                    fill: Color.YELLOW;                                 B
                } ,                                                     B
                Text                                                    C
                {   content: bind if(success)                           C
                        "SUCCESS!" else "";                             C
                    textOrigin: TextOrigin.TOP;                         C
                    fill: Color.YELLOW;                                 C
                }                                                       C
            ];
        };
    }

    package function increment() : Void                                 D
    {   if(not success)  score++;                                       D
    }                                                                   D
}
```

A Static text: "Moves:"
B Dynamic text: score
C Success message
D Increment score, if not success

The score panel is used in the lower middle part of the game display. It shows the moves taken, and a message if the player manages to reach the winning map cell. We can see it in action in figure 9.8, below.

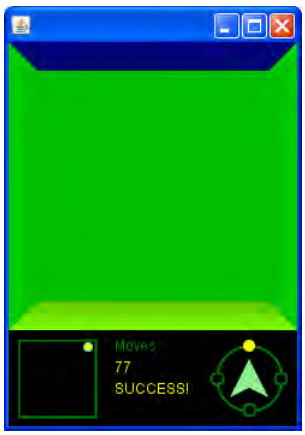


Figure 9.8 The score board sits at the bottom of the display, showing the score (moves) and a "SUCCESS!" message once the end of the maze is reached.

There's not a lot to mention about the scene graph; three `Text` nodes stacked vertically. The `increment()` function will add to the score, but only if the `success` flag is not been set. As we'll see when we check out the application class, `success` is set to true once the end of the maze if reached, and never unset. This prevents the score from rising once the maze has been solved.

9.2.6 The MazeGame class: our application

At last we get to our application's main class, where we pull the four custom nodes above together into our game's user interface. This is our application class (listing 9.10), and thanks to all the work we did with the custom nodes, it looks pretty compact and simple.

Listing 9.10 MazeGame.fx

```
package jfxia.chapter9;

import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.scene.input.KeyCode;
import javafx.scene.input.KeyEvent;
import javafx.stage.Stage;

def map:Map = Map{};
var mapDisp:MapDisplay;
var scBoard:ScoreBoard;
var comp:Compass;
var gp:Group;

Stage
{   scene: Scene
    {   content: gp = Group
        {   content:
            [   mapDisp = MapDisplay                                    A
                {   map: map;                                           A
                } ,                                                     A
                Radar                                                   B
                {   map: map;                                           B
                    translateY: 240;                                    B
                } ,                                                     B
                scBoard = ScoreBoard                                    C
                {   translateX: 88;                                     C
                    translateY: 248;                                    C
                } ,                                                     C
                comp = Compass                                          D
                {   map: map;                                           D
                    translateX: 168;                                    D
                    translateY: 248;                                    D
                }                                                       D
            ];
            onKeyPressed: keyHandler;                                   E
        }
```

```
        width: 240;  height: 320;
        fill: javafx.scene.paint.Color.BLACK;                          F
    }
}
gp.requestFocus();                                                     G

function keyHandler(ev:KeyEvent) : Void
{   def c = ev.code;                                                   H
    def x:Integer =
        if(c==KeyCode.VK_LEFT) -1                                      I
        else if(c==KeyCode.VK_RIGHT) 1                                 I
        else 0;                                                        I
    def y:Integer =
        if(c==KeyCode.VK_UP) -1                                        J
        else if(c==KeyCode.VK_DOWN) 1                                  J
        else 0;                                                        J
    def t:Integer =
        if(c==KeyCode.VK_SOFTKEY_0 or                                  K
            c==KeyCode.VK_OPEN_BRACKET) -1                             K
        else if(c==KeyCode.VK_SOFTKEY_1 or                             K
            c==KeyCode.VK_CLOSE_BRACKET) 1                             K
        else 0;                                                        K

    if(x==0 and y==0 and t==0)  return;                                L

    if( map.moveRel(x,y,t) )                                           M
    {   mapDisp.update();  comp.update();                              M
        if(t==0)  scBoard.increment();                                 M
    }                                                                  M

    if(map.success)  scBoard.success=true;                             N
}
```

A 3D map display
B Radar, lower left
C ScoreBoard, lower middle
D Compass, lower right
E Install keyboard handler
F Default background is white
G Request keyboard focus
H Which key was pressed?
I Move left or right?
J Move forward or back?
K Turn left or right?
L Nothing to do? Exit!
M Perform movement or turn
N Have we won?

After defining a few variables, we move straight into the scene graph. As this is a top level application class, we use a `Stage` and a `Scene`, then plug our custom nodes into it, via a `Group`. The `MazeDisplay` is added first, followed by the `Radar` positioned into the bottom left hand corner. Next to the radar is the `ScoreBoard`, and finally the `Compass` takes up the bottom right hand corner.

The reason we used a `Group`, rather than plug the various custom nodes directly into the `Scene` is so we can assign a keyboard handler function. In this case it's a function by the

name of `keyHandler` (what else!?), defined outside the graph for readability sake. To make the handler work we need to request the keyboard input focus, which is the purpose of the `gp.requestFocus()` call.

But what of the handler itself? The `KeyEvent` parameter contains all manner of information about the state of the keyboard when the event was generate. In this case we just want the raw code of the key which got pressed, which we copy into a handy variable. We compare this variable against the constants in the `KeyCode` class, to populate three more variables: `x`, `y` and `t`, such that `x` is -1 if moving left and 1 if moving right, `y` is -1 if moving forward and 1 if moving back, and `t` is -1 if turning left (anticlockwise) and 1 if turning right (clockwise).

You'll perhaps recognize these values as the relative movements the `Map` class accepts to make a movement; which is precisely what we do using `map.moveRel()`. The function returns true if the player successfully moved or turned, which causes us to update the `MazeDisplay` and the `Compass`. If the action was a move (not a turn) we also increment the 'moves' counter in the `ScoreBoard` class.

Finally we check to see if the winning cell has been reached, and set the `ScoreBoard`'s `success` flag if it has. Note it never gets unset, this is so the score won't rise if the player continues to move once the maze has been solved.

And that's our game complete. Now to try it out.

9.2.7 Running the MazeGame project

We can run the game in the usual way, and navigate around in glorious 3D using the keyboard's cursor keys (the arrow keys), and the square bracket keys for turning. Without cheating by looking at the data see if you can navigate your way to the end, using only the radar and compass to aid you.

Take a look back at figures 9.7 and 9.8 to see how it looks in action.

The game works perfectly well on the desktop, but our ultimate goal is to transfer it onto a mobile device emulator. If you're wondering quite how much we'll have to change to achieve that aim, then the next section might come as a pleasant surprise.

9.3 On the move: desktop to mobile, in a single bound

It's time to move our application onto a cell phone. Or, more specifically, an emulator which simulates the limited environment of a cell phone (see figure 9.9).

Figure 9.9 Our maze game hits the small screen. More specifically, it's running on the JavaFX Mobile emulator.

In the source code for the `MazeGame` class (listing 9.10) you may have noticed each action of turning, either clockwise or anticlockwise, was wired to a couple of keys. The `KeyCode.VK_SOFTKEY_0` and the `KeyCode.VK_SOFTKEY_1` could be used in preference to `KeyCode.VK_OPEN_BRACKET` and `KeyCode.VK_CLOSE_BRACKET`. As you may have guessed, this is to accommodate the limited key input on many mobile devices.

This is really the only concession I had to make for the mobile environment. The rest of the code is mobile-ready. When writing the game I was very careful to only use those classes available for the *common profile*.

"The common what?!?"

The common profile are those classes which are available across the whole range of JavaFX environments, from desktop, to applet, to mobile and TV. Classes in JavaFX are assigned to one or more profiles. Classes which can only be used on the desktop, for example, are in the *desktop profile*. Similarly classes which may only be available on JavaFX TV are exclusive to the TV profile (hypothetically speaking, given that the TV platform is months away from release as I write this). Classes which span all environments, devices and platforms are said to be in the *common profile*. By sticking to only that part of the JavaFX API which is in the common profile, we ensure the maze game will work on the phone emulator.

The official JavaFX web documentation features a toggle (in the form of a group of links at the head of each page) for showing only those bits of code inside a given profile. This is handy when we wish to know the devices and platforms our finished code will be able to run on. Flip the switch to "common" and all the desktop only classes will vanish from the page, leaving only those safe to use for mobile development.

So, we don't need any changes to the software, our game is already primed to go mobile. We just need to know how to get it there!

9.3.1 Packaging the game for the mobile profile

To get the game ready for the emulator we use the trusty `javapackager` tool, first encountered in the Enigma chapter to bundle up code ready for the web. In this scenario, however, we want to output something suitable for a mobile phone; but fear not, the packager can do that too.

```
javafxpackager -profile MOBILE -src .\src
  -appClass jfxia.chapter9.MazeGame
  -appWidth 240 -appHeight 320
```

The above is the command line we need to call the packager with, split over several lines. If you're on a Unix flavored machine, the directory slashes run the opposite way, like so:

```
javafxpackager -profile MOBILE -src ./src
  -appClass jfxia.chapter9.MazeGame
  -appWidth 240 -appHeight 320
```

The command assumes you are sitting in the project's directory, with `src` immediately off from your current directory. The options for the packager are quite straight forward. The application's source directory, its main class name, and its dimensions, should all be familiar. The only change from when we used the packager to create an applet is the `-profile MOBILE` option. This time we're asking to output to the mobile environment, instead of the desktop. (If you need to remind yourself of how the packager works, take a look back at section 8.4.2, last chapter.)

A `dist` directory is created within our project, and inside you should find two files:

- The `MazeGame.jar` file is the game code, as you would expect.
- Its companion, `MazeGame.jad`, is a definition file full of meta data, helping mobile devices to download the code, and users to know what it does before they download it.

Going mobile, with NetBeans

Once again I'm showing you how things happen under the hood, without the abstraction of any given IDE. If you're running NetBeans you should be able to build and package the mobile application without leaving your development environment. Terrence Barr has

written an interesting blog entry, explaining how to create and build mobile projects from NetBeans (the following web address has been broken over two lines).

```
http://weblogs.java.net/blog/terrencebarr/
  archive/2008/12/javafx_10_is_he.html
```

The JAR and the JAD file are all we need to get our application onto a phone. So, given we don't have an actual physical phone to test the game on, the final step is to test it in the emulator.

9.3.2 Running the mobile emulator

The mobile emulator is a simulation of the hardware and software environment in typical mobile phones. The emulator enables us to test and debug a new application without using an actual cell phone.

You'll find the emulator inside the `emulator\bin` directory of your JavaFX installation. On my Windows XP box, with a standard JavaFX SDK 1.0 install, this equates to `"C:\Program Files\JavaFX\javafx-sdk1.0\emulator\bin\emulator.exe"`. To run the emulator with our maze game, we merely need to point its `-Xdescriptor` option at our JAD file, like so:

```
"C:\Program Files\JavaFX\javafx-sdk1.0\emulator\bin\emulator"
  -Xdescriptor:dist\MazeGame.jad
```

When you run this command, a little service program will start up called the “JavaFX SDK 1.0 Device Manager”. You'll see its icon in the system tray (typically located in the south-east corner of the screen on Windows). If you are running a firewall (and I sincerely hope you are, if you're connected) you may get a few alerts at this point, asking you to sanction connections from this new software. Providing the details relate to Java (and not some other mysterious application) you should grant the required permissions, allowing the Device Manager and emulator to run. The result should look like figure 9.10.

Figure 9.10 The mobile emulator in action (or at least, its top two thirds!)

Once the emulator has fired up, you can use its navigation buttons, and the two function buttons directly below the screen, to move around. Clicking the navigation buttons moves us around the maze, while the function buttons are for turning.

And that's it, our maze has successfully gone mobile!

Running the emulator, with NetBeans

If you want to remain within your IDE, and still run the emulator, there's a guide to working with the emulator from NetBeans under the section entitled "Using the Run in Mobile Emulator Execution Model" at the web address below (broken over two lines).

```
http://javafx.com/docs/tutorials/
  deployment/configure-for-deploy.jsp
```

9.4 Summary

In this chapter we've pushed the scene graph further than anything we've seen before, traveling beyond the flat world of two dimensions. We also got a small taster of what's to come when the JavaFX Mobile profile is released (which, hopefully, may be by the time you read this book).

As always with the projects in this book, there's still plenty of room for experimentation. For example, the game works fine on 240x320 resolution displays, but needs to be scaled and rearranged for alternative screen sizes. (Hint: the `MazeDisplay` class supports a scaling factor, which you might want to investigate.) The radar, compass and scoreboard also seem to exhibit a few graphical quirks when running in the preview release emulator: you may want to experiment with soothing these.

This chapter went through quite a journey during the writing of the book, some of which is worth mentioning from a technical point of view. Ear-marked originally as a mobile chapter, by the Fall of 2008 it looked increasingly uncertain whether mobile support would be in the initial JavaFX release. The first draft was targeted at the desktop, therefore, and featured a far more complex scene graph which applied perspective effects onto bitmap images, overlaid with translucent polygons to provide 'darkness'. Very 'Dungeon Master'! (See figure 9.11.) However, when JavaFX 1.0 arrived in December complete with bonus *Mobile* preview, the code was stripped back radically to become more suitable for a mobile device.

Figure 9.11 A desktop version of the 3D maze, complete with bitmap walls using a perspective effect. Sadly the bitmaps had to go when the project was adapted to fit a mobile platform.

Originally I wanted to manufacture the 3D scene using SVG. Create a vector based wall image, then replicate it for all the various sizes and distortions needed to form the 19 wall pieces. Each piece would be on a different ("`jfx:`" labeled) layer inside a single SVG, and could be switched on or off independently to show or hide the wall nodes. A little bit of extra

effort up front with Inkscape (or Illiustrator) would produce much cleaner source code, effectively removing the need for `_wallFront()` and `_wallSides()`. We could even have different looking wall blocks, using a collection of prepared FXZ files. Imagine my disappointment, then, when I discovered the FXD library wasn't yet compatible with the Mobile profile. Oh well... hopefully the situation will have improved by the time you read this.

So it's early days for the Mobile profile, clearly; but even the preview release used for this chapter shows great promise. As mobile devices jump in performance, and their multimedia prowess increases, the desktop and the hand-held spaces are slowly converging. JavaFX allows us to take advantage of this trend, targeting multiple environments *in a single bound* through a common profile, rather than coding from scratch for every platform our application is delivered on.

So thus far in the book we've used JavaFX on the desktop, taken a short hop over to a web browser, then a mighty leap onto a phone, all without breaking a sweat. Is there anywhere else JavaFX can go? Well, yes!, it can also run *inside* other applications, executing scripts and creating bits of UI. Although a rather esoteric skill, it can (in very particular circumstances) be incredibly useful. So that's where we'll be heading next.

10

Best of both worlds: using JavaFX from Java

In previous chapters we saw plenty of example demonstrating Java classes being used from JavaFX Script. Now its time for an about face; how do we call JavaFX Script from Java?

As JavaFX Script compiles directly to JRE compatible bytecode, it might be tempting to assume we can treat JFX 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 about JavaFX Script classes to be a dangerous business. For example, JavaFX Script classes have no formal constructor, as such, so any constructors in the bytecode are the result of compiler implementation detail–we can't guarantee future JavaFX Script compilers will work the same way. So, in this chapter we'll look at how to get the two languages talking in a manner which won't leave any ticking time bombs behind.

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 which relies on a huge JAR library written in Java, is my program a Java program which uses JavaFX Script, or a JavaFX Script program which 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 which forms the entry point into the application. In the above example it's JavaFX Script which runs first, placing our hypothetical application unambiguously into the category of a JavaFX application which 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.

10.1 Different styles of linking the two languages

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

1. 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 user interface in it. In this scenario JavaFX Script would be merely standing in for Java in some parts of our application.
2. As a runtime scripting language. We might want to add scripting capabilities to our Java application, and decide JavaFX Script is a suitable language to achieve this. In this scenario JavaFX Script is acting as an end user tool; a means for users to manipulate our application in a programmatic way.

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

10.2 Adventures in JavaFX Script

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

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 more lucky, a ready-made palette of isometric game tiles (painstakingly crafted in Deluxe Paint, I seem to recall) survived on an old hard drive (see figure 10.1) 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!

Figure 10.1 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.

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. As 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, apart from a bare bones description. The game engine is just a means to an end—a sample application we can use as a test bed for our JavaFX Script integration. We'll be focusing 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 web site. 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/morris/

10.2.1 How our adventure game engine works

For the sake of flexibility, many games are developed in two distinct parts. A programmer will build a *game engine*, which drives the game based upon the graphics, sounds and level or 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 upon with the Enigma applet project.

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 10.2 for a visual representation of how they are arranged. Each cell can have a floor tile, two wall blocks, and two *face* tiles. Face tiles are used to add quirky detail onto a wall, or objects like torches or switches onto the wall surface.

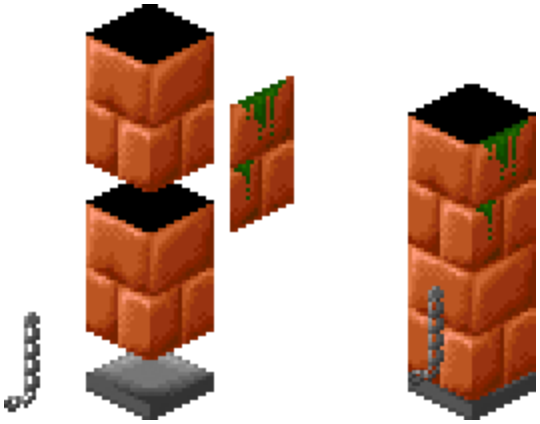


Figure 10.2 Each cell in the game environment is created from up to five images, one floor tile, two wall tiles, and two faces which modify one side of a wall tile.

Unlike with the maze we developed previously, the game map isn't hard coded 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.

10.2.2 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 hard coding details into the game code itself. Our game engine uses JavaFX Script as its scripting engine, to support three types of game event.

1. An event can run whenever the player leaves a given cell.
2. An event can run whenever the player enters a given cell.
3. We can define *actions*, which the player triggers by standing on a given cell and pressing 'space' 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 which appear in each cell, and the door links which move the player between rooms. More importantly, it also includes the event code. We can see how a part of it might look in listing 10.1, below.

Listing 10.1 A fragment of the game data file

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

#SCRIPT 1 4 1 action                                                        C
if(state.getPlayerFacing()==0)                                              C
{   // Flip switch                                                          C
    state.setRoomCell(1,4,0 , -1,-1,-1,0x4e,-1);                            C
    // Remove barrel in opening room                                        C
    state.setRoomCell(0,6,1 , -1,0,-1,-1,-1);                               C
}                                                                           C
#END                                                                        C
```

A Grid of tile graphics
B Door link
C Action event as JavaFX Script

We only have a fragment of the data file on show here, detailing the code for just one room. The data file, in its entirety, is loaded by the Java game engine when it first starts up. The format is one I made up myself, and serves to quickly create a game for testing our engine. If I was coding the engine for real I might invest time in devising something like a full XML format.

The #ROOM block at the head of the code describes the tiles and title of the room, which is defined as room number 1 and 6x6 cells in size. Next we have a #LINK connecting room 1, cell (5,1) facing east, with room 0, cell (1,2) facing east. The #REPAINT line is a series of boolean flags, one for each cell, which helps the engine optimize screen updates. Finally we have the #SCRIPT block we're interested in, attached to room 1, cell (4,1) as an *action* event type. Recall, the action event type means it will run when the player presses the space key.

Figure 10.3 shows the room in question. The event script in listing 10.1 checks to see whether the player is facing north (zero means north), and if so changes a tile at cell (4,0), which flips the switch mounted on the wall into the up position. It also changes a tile in another room, which helps the player progress further in the game.

Figure 10.3 This is room 1, which the fragment of data file in listing 10.1 refers to. The player is standing on cell (3,1), in front of him is the event cell (4,1), and beyond that the door link cell (5,1).

The script does all of these things by calling functions on an object named `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 which calls our script. To investigate further we need to check out that Java code, which is precisely where we'll head next.

10.2.3 Calling the JavaFX Script event code from Java

Java SE version 6 includes JSR 223. In plain English: the Java Specification Request for calling scripting languages from within Java programs (aka JSR 223) is part of the Standard Edition of Java 6. Providing a scripting language implements the necessary *engine* (not to be confused with game engines, mentioned above) it becomes possible for Java to run and interact with scripts written in that language.

JSR 223 uses what's known as the *service provider mechanism*, meaning all we have to do is include the classes which implement the scripting engine on Java's classpath, and the engine will be found when requested. Fortunately, even though we generally work with JavaFX Script as a compiled language, JavaFX comes with the required engine to treat it as a scripting language. It lives in the `javafxc.jar` file, inside the `lib` directory of your JavaFX SDK installation. It utilize it all we have to do is add the JAR to our classpath.

Listing 10.2 shows fragments (small snippets, from various points in the code) of the `Map` class, part of the Java game engine I implemented for this project. It's part one of the listing, with the second part to follow.

Listing 10.2 Map.java (part 1) : calling JavaFX Script from Java

```
// Head of file
import javax.script.ScriptEngineManager;                                A
import com.sun.javafx.api.JavaFXScriptEngine;                           A

// ...

// Class variables
private JavaFXScriptEngine jfxScriptEngine;                             B

// ...

// Inside the constructor
ScriptEngineManager manager = new ScriptEngineManager();                C
jfxScriptEngine = (JavaFXScriptEngine)                                  C
    manager.getEngineByName("javafx");                                  C

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

A Import JavaFX scripting engine
B Reference to JFX engine
C Setup engine
D Add Java 'state' object

At the head of the listing we have the imports required to allow 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.

A class variable references the engine once we've discovered it. Then, in the class constructor, we use Java's `ScriptEngineManager` to get a reference to JavaFX's `JavaFXScriptEngine`. You can see that we first create a new manager, and then ask it to find (using the service provider mechanism) a scripting engine which matches the token "`javafx`". Assuming the necessary JavaFX Script JAR file is on the class path, the manager should get a positive match.

Having acquired a reference to a JavaFX Script scripting engine, we add a Java object, `state`, into the engine's runtime environment. This is the variable we encountered in the last section. 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 the `state` object is shared with the JavaFX Script environment. We can share as many Java objects as we please like this, and choose what names the appear under in the script environment (in our code, however, we stick with "`state`").

Our scripting engine is all set up and ready to use. In part two of the code (listing 10.3) we look at how to call it.

Listing 10.3 Map.java (part 2) : calling JavaFX Script from Java

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

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

    // Perform script
    int key = kx*100+ky;
    if(hash.containsKey(key))
    {   String script = hash.get(key);                                   C
        try                                                              D
        {   jfxScriptEngine.eval(script);                                D
        }catch(Exception e) { e.printStackTrace(); }                     D
    }

    // True if player moved
    return !(state.playerX==x && state.playerY==y &&
        state.playerFacing==f && currentRoom==r);
}
```

A Three different event types
B Handle each game event
C Event code from game data
D Execute code

The three types of events our engine supports each have their own method. The game engine uses three hash tables to store the events. Each of these methods 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 the raw JavaFX Script source code we saw in the data file earlier. To run this code we use the JavaFX Script interpreter we created as a JSR 223 scripting engine, and that's precisely what happens next. The call to `eval()`, passing the JavaFX Script code, causes it to run. An exception handler is needed around the `eval()` call, to catch runtime errors thrown by the script.

Important: Getting the classpath right

When you run a JavaFX program, using the `javafx` command (directly, or via an IDE), the standard JFX 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 depends upon which features of JavaFX you use, but typically you want to include `shared/javafxrt.jar`, `desktop/javafxgui.jar`, `desktop/javafx-swing.jar` and `desktop/Scenario.jar`. If you're using the JavaFX Script JSR 223 scripting engine, you'll need to include `shared/javafxc.jar` as well. This is the JAR containing the compiler and the JSR 223 engine code.

10.3 Adding a little FX to Java

So far we've looked at how to treat JavaFX Script as a runtime scripting language. But, 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 JavaFX and AWT/Swing. Sadly this class was removed from the full release, making it impossible to pass anything except the Swing wrapper nodes back to a Java UI. Even so, there are a couple of reason why it is still useful to demonstrate this technique:

- It's entirely possible the JavaFX team may add back in a bridge between the JavaFX scene graph and Java AWT/Swing in a later release.
- Just because JFX nodes aren't compatible with Swing, doesn't mean this technique cannot be used to build other types of data structure using JavaFX Script's declarative syntax, and hand them back to Java.

JavaFX scene graph inside Java: the hack

Sun engineer Josh Marinacci posted a quick-n-dirty hack from fellow Java/JFX team members Richard Bair and Jasper Potts, building a bridge between the JavaFX (release 1.0) scene graph and Swing. The code carries a massive “this is a hack” health warning, and is unlikely to work in later releases, but for what it's worth you can read about it here:

`http://blogs.sun.com/javafx/entry/how_to_use_javafx_in`

As an example our project will use a very basic control panel below the main game view. The simple panel is actually a JavaFX Script created `SwingLabel`. To see how it looks, take a glance at figure 10.4, below.

Figure 10.4 The panel at the foot of the game's window is written entirely in compiled JavaFX Script.

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

10.3.1 The problem with mixing languages

Stop and think for a moment about just what we're trying to do here; we have three languages in play: the Java language, the JavaFX Script language, and the bytecode both of them are compiled to.

When we used JSR 223, the scripting engine took raw JavaFX Script source code and ran it for us. Although Java and JavaFX Script were interacting, they were doing so through the high level abstraction of the scripting engine. It wasn't as if we had two sets of bytecode files, compiled from two different languages, being merged into one application.

JavaFX Script guarantees a high degree of interoperability with Java as part of its language syntax; as such we don't have to worry about how Java code is turned into runnable bytecode classes. However the same is not true when we travel in the opposite direction. For example, although it seems likely that JavaFX Script functions will be translated directly into bytecode methods, this is an assumption we really shouldn't make if we want to be 100% sure of not tripping ourselves up with future (or rival) JavaFX Script compilers. So, how do we link the two languages without making any such assumptions?

Fortunately an elegant solution emerges after a little bit of lateral thinking. We can't rely on the relationship when Java uses JavaFX Script classes, but we *can* rely on the relationship when JavaFX Script uses Java classes. We can use the latter to fix the former.

How? Java interfaces, that's how!

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

If we encode the interactions between our two languages into a Java interface, then 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 which Java can exploit to communicate with the JavaFX Script software.

Skinning cats...

We can also form a bridge by subclassing a Java class too. 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 which makes sense to you, and your current project.

Listing 10.4 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 Swing compatible object (a `JComponent`), and the second is used to interact with user interface.

Listing 10.4 ControlPanel.java

```
package jfxia.chapter10;

import javax.swing.JComponent;

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

A Fetch Java compatible UI
B Interact with UI

Right, now we're familiar with the interface and its methods, let's see them in action. Listing 10.5 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 10.5 Game.java (part 1) : adding JavaFX user interfaces to Java code

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

// ...

// Class variables
private ControlPanel ctrlPan;                                     B

// ...

// Initialize the user interface
ctrlPan = getJavaFX();                                            C
ctrlPan.setTitle(state.getCurrentRoomTitle());                    C

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

A Imports required
B Control panel reference
C Create and setup
D Hook into Java Swing

The imports at the head of the file should be familiar from listing 10.2, earlier. We create a class variable to hold our JavaFX object reference, using the `ControlPanel` interface we defined as a bridge. Finally, the meaty part: the control panel is plugged into the Swing UI.

To plug the two user interfaces together we use the method `getJavaFX()`, which fetches the `ControlPanel` (details in listing part two, be patient!) Remember, the `ControlPanel` class isn't a UI component itself, but a bridge between the Java and the JavaFX Script. 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.

10.3.3 Fetching the JavaFX Script object from within Java

The code which creates the `ControlPanel` class will look very familiar. It's just a variation on the JSR 223 scripting engine code we saw earlier in listing 10.3.

Listing 10.6 is the second half of our `Game` class fragments. It shows the method `getJavaFX()` using the JavaFX Script scripting engine.

Listing 10.6 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.chapter10.jfx.ControlPanelImpl;\n"+
            "return ControlPanelImpl{};"
        );
    }
    catch(Exception e)
    {   e.printStackTrace();
        return null;
    }
}
```

As we're only going to use the scripting engine 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 10.5 demonstrates the full relationship. Once the `ControlPanelImpl` object is created, the 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 direct support for Java compatible constructors means the `ControlPanelImpl` object must initially be created using JSR 233.

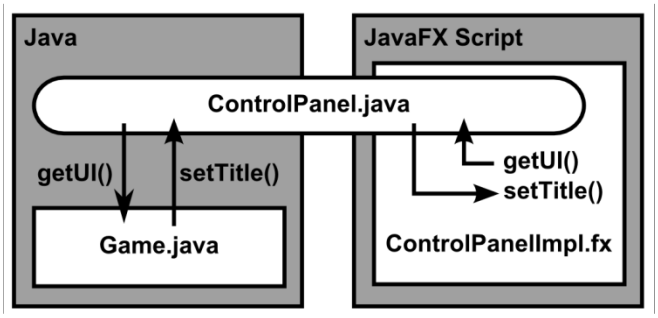


Figure10.5 Java's Game class and the JavaFX Script ControlPanelImpl.fx class communicate via a Java interface, ControlPanel.java

Let's take a look at `ControlPanelImpl` right now. As it's entirely written in JavaFX Script, the code is presented in its entirety below in listing 10.7 (for those suffering JFX withdrawal symptoms, after such a Java-heavy chapter).

Listing 10.7 ControlPanelImpl.fx

```
package jfxia.chapter10.jfx;

// These are required for our CustomNode
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;                                          A
import jfxia.chapter10.ControlPanel;                                    A


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

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

```
        }
    }

    public override function getUI() : JComponent                          D
    {   myNode.getJComponent();                                            D
    }                                                                      D
                                                                           D
    public override function setTitle(s:String):Void                       D
    {   text = s;                                                          D
    }                                                                      D
}
```

A Required for Java coupling
B Subclass our Java interface
C Create and store node
D ControlPanel interface methods/functions

Our final piece in the puzzle! Ignoring the `ControlPanel` interface sections at the foot, the script is a pretty standard Swing node. At the foot of the listing we see two functions which 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 just a minor point, but an important one: the JavaFX Script class relies upon the Java interface, so whatever process or IDE we use to build the source code, we need to ensure the Java is compiled first. The Java doesn't need the interface implementation to be available when it builds; the whole idea of interfaces is the implementation can be plugged in later. However the JavaFX Script 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 only have to re-compile the Java, leaving the JavaFX Script classes 'as is'. Likewise, the reverse is true if all the changes are restricted to the JavaFX Script side.

You may want to make use of this natural separation when you design your project, perhaps assigning the Java code and the JavaFX Script code to different teams of developers, coding independently against interfaces which bridge their work.

10.4 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 which 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 full within existing Java applications. JFX would surely give us a powerful tool in slashing development times of Java desktop apps, and reducing the frequency of UI bugs.

And so this chapter brings the book to a close. We started way back in chapter one with some excited talk about the promise of Rich Internet Applications and the power of Domain Specific Languages, we 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, send some secret codes from within a web browser, got lost in a mobile maze, and finally 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 1.0, which still is missing a lot of really useful functionality; my goal was to give you a good grounding in the concepts and ideas which surround JavaFX, so you could take future enhancements with ease. 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. Then try building your own apps. We're living in an age of sumptuous graphical effects and rich multimedia; 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!

Have fun!

Appendix A

Getting started

This section provides details on how to download, install, setup, compile and run the JavaFX development environment. It also provides links to useful JavaFX web resources.

Everything you need to get started with JavaFX is free to download, for Microsoft Windows, Mac OSX or Linux.

A1 Downloading and installing

Before we can start writing JavaFX applications we need the right software. Below you'll find web addresses to the downloads you need to get started. For each piece of software read the licenses (if any) and requirements carefully, and follow the download and installation instructions.

Please read over this A1 section before downloading anything, particularly if you're new to Java. It contains information in later sections which helps you get a solid overview of which tools you may want to download and install.

The JavaFX tools themselves fall into two categories: the “JavaFX SDK” for programmers, and the “JavaFX Production Suite” for designers. Their contents are outlined in the sections which deal with each, below. This book primarily focuses on the programmer tool (the SDK), although for the full experience you're recommended to download both.

A1.1 The Java Development Kit (essential)

Firstly, if you don't have the latest version already, you need to visit the Java Standard Edition downloads page to fetch and install a copy of the Java SE JDK. Sun provides SE (Standard Edition) implementations for Microsoft Windows and Linux, including 64 bit builds. If it's available for your system, you're strongly urged to get Java 6 JDK, Update 11 (or later), due to radical enhancements in applet deployment and Java start up times.

Remember: you need to download the JDK, not the JRE. The Java Runtime Environment does not contain the extra tools needed to write Java software, merely run it. (The JDK comes bundled with a compatible JRE).

JAVA SE DOWNLOADS (WINDOWS AND LINUX)

http://java.sun.com/javase/downloads/index.jsp

APPLE'S JAVA DEVELOPMENT KIT (MAC OS X)

http://developer.apple.com/java/

At the time of writing, for JavaFX 1.0, the requirements are:

- For Windows: Windows XP (SP2) or Vista, running JDK 6 Update 11 (recommended) or JDK 6 Update 7 (minimum).
- For Mac: Mac OS X 10.4.10 or later, running JDK 5 Update 13 (aka v1.5.0_13) (recommended), or Java for Mac OS X 10.4 Release 7 / Java for Mac OS X 10.5.4+ Update 2 (minimum).
- For Linux: Sorry, details currently unavailable.

A1.2 NetBeans or other IDEs (optional)

You'll find plenty of references to NetBeans on Sun's Java pages, including options to download the NetBeans program as a bundle with the JDK. The JDK provides the raw tools for Java, like the language compiler and runtime environment; it does not contain a source code editor, or an Integrated Development Environment. An IDE is a a seamless software environment, providing sophisticated visual tools for editing, analyzing and debugging code. Most of these tools are available in the standard JDK, but as command line (eg. 'shell' or MS-DOS) programs rather than GUI based applications.

If you're not at home with command line environments, you are strongly recommended to install an IDE to help you write your software. Modern IDEs are quite sophisticated, they can be extended with new technologies, like the JavaFX Script compiler used to build JavaFX code. Such extensions come in the form of *plugins*, installed into the IDE to teach it how to interface to new tools. NetBeans is Sun's own IDE, however it is not the only one available (nor, indeed, is it necessarily the most popular). Rivals of note include Eclipse and IntelliJ, each with its own devoted fan following. You may want to investigate these alternatives before you make a choice, but it's important to check whether the necessary plugins have been crafted for your chosen IDE to integrate into the tools you want to run; especially JavaFX.

As NetBeans is Sun's own IDE, it's plugin support for JavaFX is guaranteed, and likely to be one step ahead of its rivals. By all means investigate the various options (talk to fellow programmers and see which IDE they recommend) but if you're still uncertain, download NetBeans–it's by far the safest option.

NETBEANS

http://www.netbeans.org/

ECLIPSE

http://www.eclipse.org/

INTELLIJ

http://www.jetbrains.com/idea/ (Warning: no JFX plugin at time of writing.)

This book does not favor one IDE over any other. Each IDE has its own tutorials and documentation, as well as strong on-line fan base. Covering every IDE would place an unnecessary burden on the text, while picking only one would alienate devotees of rival platforms. Fortunately on-line tutorials are available.

BUILDING A JAVAFX APPLICATION USING NETBEANS IDE

http://java.sun.com/javafx/1/tutorials/build-javafx-nb-app/

DEVELOPING JAVAFX APPLICATIONS WITH ECLIPSE – TUTORIAL (LARS VOGEL)

http://www.vogella.de/articles/JavaFX/article.html

A1.3 The IDE plugins (required, if using an IDE)

Above we talked about IDEs and their merits. Each IDE needs a plugin to teach it how to use the JavaFX SDK tools. The links below will lead you to the relevant project page.

NETBEANS JAVAFX PLUGIN

http://javafx.netbeans.org/

ECLIPSE JAVAFX PLUGIN

http://kenai.com/projects/eplugin

INTELLIJ JAVAFX PLUGIN

No plugin at the time of writing. Check for updates at: http://plugins.intellij.net/

A1.4 The JavaFX SDK (essential)

Having downloaded and installed the latest Java SE JDK, and perhaps a favored IDE, you now need to do the same for the JavaFX Software Developers Kit. This provides the extra JFX components for creating JavaFX software to target the Java SE platform.

As well as a project web site, JavaFX has a slick marketing-focused site from where you can download the latest official release.

THE JAVAFX SITE

http://www.javafx.com/

THE JAVAFX SITE DOWNLOAD PAGE (DIRECT LINK)

http://javafx.com/downloads/all.jsp

The OpenJFX project is an Open Source initiative, sponsored by Sun Microsystems, to create the JavaFX Script compiler (and presumably, in future, the API as well). The project site contains the latest compiler releases and source code. Only download these if you want the bleeding-edge alpha or beta versions, with all the in-progress fixes and additions.

THE OPENJFX PROJECT HOME PAGE

https://openjfx.dev.java.net/

THE JAVAFX BUILD SERVER

http://openjfx.java.sun.com/

A1.5 The JavaFX Production Suite (optional)

The JavaFX Production Suite, previously known under its codename of Project Nile, is a set of plugins for both Adobe Illustrator (CS3+) and Adobe Photoshop (CS3+), allowing export into a special JavaFX graphics format. There's also a SVG (Scalable Vector Graphics) converter, and a viewer for the JavaFX format.

JAVAFX PRODUCTION SUITE DOWNLOAD

http://javafx.com/downloads/all.jsp

GETTING STARTED WITH JAVAFX 1.0 PRODUCTION SUITE

http://javafx.com/docs/gettingstarted/production_suite/

The Production Suite is aimed at designers; you don't need it if just writing software. But if you own the CS3 versions of Illustrator or Photoshop, or you fancy converting SVG images into a format more closely relating to the JFX scene graph, then why not give it a try?

Readers unfortunate enough not to own said Adobe applications might be interested in a highly regarded Open Source vector graphics program, by the name of Inkscape.

INKSCAPE

http://www.inkscape.org/

INKSCAPE AND JAVAFX WORKING TOGETHER (SILVEIRA NETO)

http://silveiraneto.net/2008/11/21/inkscape-and-javafx-working-together/

A1.6 Recap

Just to sum up the previous sections, the minimum you must have installed to begin JavaFX development is:

- The Java SE Development Kit (JDK).
- The JavaFX SDK.

Optionally you may also wish to install:

- An IDE of your choice, with its associated JavaFX plugin (if the command line isn't your *thing*).
- The JavaFX Production Suite.
- Inkscape (as an alternative to Adobe Illustrator).

And if you're the kind of expert programmer who wants to keep their finger on the pulse of the JavaFX Script compiler:

- The latest OpenJFX build.

A2 Compiling JavaFX

Having downloaded and installed the latest Java Development Kit, and a JavaFX SDK, you'll need to know how to run the JavaFX Script compiler, and execute the programs it creates. If you plan on developing all your software exclusively from within an IDE you don't need to pay too much attention to this section (although it's always handy to know what's happening under the hood).

I'm going to explain the necessary steps in a platform-agnostic way. It is assumed you're already familiar with the ins and outs of your computer's command line interface, be that something like Bash on Unix flavored platforms, Microsoft's own MS-DOS, or whatever CLI your system provides. Readers who aren't compliment with a command line are strongly recommended to use an IDE..

A2.1 Setting the paths

To get access to all the tools you needs, both the JDK and the JavaFX SDK need to be in your path.

Take a look at the directory where your Java Development Kit was installed. Inside you'll find various directories, one of which should be called `bin`. This is the directory with all the Java tools in, and it needs to be on your execution path.

Then take a look at where you installed the JavaFX SDK. Again, you'll various directories inside its installation directory, one of which will also be called `bin`. This houses all the JavaFX tools, and it too needs to be on your execution path.

To text that both these directories are correctly set on your path, type the following at the command line:

```
javac -version
javafxc -version
```

If the paths are correct each command will cause the respective compiler to run, printing out its version information. Next we need to check the runtime environments are set. Type the following into the command line:

```
java -version
javafx -version
```

If the compiler commands worked then it's very unlikely the above runtime commands will fail. However, **make sure the versions are the ones you expect!** Some operating systems come pre-installed with Java Runtime Environment's, sometimes woefully out of date, so it's always a good idea to double check you have the right runtime on your path.

The `javafx` command is actually a convenience wrapper around the `java` command, and so both commands return the Java version information.

A2.2 Running the compiler

Having set up the tools on the path, building software is merely a case of calling the compiler and pointing it at your source files. The JavaFX Script compiler takes most of the same command line options as the Java compiler. You can get a listing of them with the command:

```
javafxc -help
```

A simple compilation, on Windows or Unix, might look like this:

```
javafxc -d . src\*.fx          (Windows/MS-DOS)
javafxc -d . src/*.fx          (Unix variant)
```

The compiler will run building all the source files inside the `src` directory, outputting its classes into the current directory. If the classes use packages, you'll find the output directory will contain a directory structure mirroring the package structure. For more elaborate control over the build process you might want to investigate the Apache Ant project. Ant is an Open Source initiative which allows the programmer to describe (using an XML file) the steps and dependencies necessary to build a program. Newer versions of Ant have support for calling the JavaFX Script compiler as part of their build process.

APACHE ANT

http://ant.apache.org/

A2.3 Running the code

Having built our code, the final step is to run it. The JavaFX runtime command wraps the Java runtime command, adding the extra JavaFX API classes onto the class path for us. A list of options can be displayed using the following command:

```
javafx -help
```

A typical invocation of the JavaFX runtime might look something like this:

```
javafx -cp . mypackage.MyClass
```

The class `mypackage.MyClass` is loaded and run, assuming it lives in the current working directory. The `-cp` option adds the current directory onto the class path, ensuring our class will be found by the JRE.

A3 Useful links

Learning a new platform is always hard at first. It's nice to have resources you can turn to for guidance, information and inspiration. This section lists web links which may be useful to you in your JavaFX work. The links are loosely grouped by type, but are in no particular order of preference.

BOOK: JAVAFX IN ACTION, WEBSITE

http://www.manning.com/morris/

JAVAFX: THE JAVAFX SITE

http://www.javafx.com/

JAVAFX: JAVAFX TECHNOLOGY AT A GLANCE

http://java.sun.com/javafx/index.jsp

JAVAFX: JAVAFX 1.0 API DOCUMENTATION

http://java.sun.com/javafx/1/docs/api/

JAVAFX: JAVAFX SCRIPT 1.0 LANGUAGE REFERENCE

http://openjfx.java.sun.com/current-build/doc/reference/JavaFXReference.html

JAVAFX: THE OPENJFX PROJECT HOME PAGE

https://openjfx.dev.java.net/

JAVA: SUN'S JAVA DEVELOPER HOME PAGE

http://java.sun.com/

COMMUNITY: SUN'S JAVA COMMUNITY WEB SITE

http://java.net/

COMMUNITY: JAVALOBBY DEVELOPER COMMUNITY

http://java.dzone.com/

BLOG: JAMES WEAVER'S JAVAFX BLOG

http://learnjavafx.typepad.com/

BLOG: CHRIS OLIVER'S BLOG

http://blogs.sun.com/chrisoliver/

BLOG: JOSHUA MARINACCI'S BLOG

http://weblogs.java.net/blog/joshy/

Appendix B

JavaFX Script: A very quick reference

This appendix is an ultra-terse guide to JavaFX Script, almost like *flash cards* for each of the topics covered in the main language tutorial chapters, earlier in this book. As well as acting as an quick reference to the language syntax, it could also be used by highly confident developers (impatient to get on to the projects) as a fast-track to learning the basics of the JavaFX Script language.

B1 Comments

JavaFX Script supports comments in the same way as Java:

```
// A single line comment.
/* A multi line comment. */
```

B2 Variables and data types – the basics

JavaFX Script variables are statically typed, not dynamically typed. The language has no concept of primitives in the way Java does, everything is an object. Some types, however, may be declared using a literal syntax. We call these the *value types*, and they are: `Boolean`, `Duration`, `Integer`, `Number` and `String`. `KeyFrame`, an animation class, also has its own specific literal syntax (dealt with in the chapters dealing with animation time lines).

Aside from having their own literal syntax, value types can never be `null`, meaning unassigned value types have default values. Value types are also immutable.

B2.1 Variable declaration (def, var, Boolean, Integer, Number, String)

Value types are created using the `var` or the `def` keyword, followed by the variable's name, and optionally a colon and a type.

```
var valBool:Boolean = true;
var valInt:Integer = 8;
```

```
var valNum:Number = 1.245;
var valStr:String = "Example text";
```

If initial values are not provided, sensible defaults are used. JavaFX Script also supports duck typing.

```
var assBool = true;
var assInt = 1;
var assNum = 1.1;
var assStr = "Some text";
```

The `def` keyword prevents a variable from being reassigned, once its initial value is set. This aids readability, bug detection, and performance. Note: although the variable cannot be reassigned, if its body uses a one-way bind, its value can still change.

```
var canAssign:Integer = 5;
def cannotAssign:Integer = 5;
canAssign = 55;
cannotAssign = 55;  // Compiler error
```

B2.2 Arithmetic (+, -, etc.)

Numbers may be manipulated in mostly the same ways as Java when it comes to arithmetic. Remember though, all variables are objects, and so are literals in the source code.

```
def n1:Number = 1.5;
def n2:Number = 2.0;
var nAdd = n1 + n2;
var nSub = n1 - n2;
var nMul = n1 * n2;
var nDiv = n1 / n2;
var iNum = n1.intValue();

var int1 = 10;
var int2 = 10;
int1 *= 2;
int2 *= 5;
var int3 = 9 mod (4+2*2);
var num:Number = 1.0/(2.5).intValue();

def dec = 16;      // 16 in decimal
def hex = 0x10;    // 16 in hexadecimal
def oct = 020;     // 16 in octal
```

B2.3 Logic operators (and, or, <, >, =, >=, <=, !=)

Comparisons between variables are done pretty much the same way as in Java, except the `&&` and `||` symbols are replaced by the keywords `and` and `or`.

```
def testVal = 99;
var flag1 = (testVal == 99);
var flag2 = (testVal != 99);
var flag3 = (testVal <= 100);
var flag4 = (flag1 or flag2);
var flag5 = (flag1 and flag2);
var today:java.util.Date = new java.util.Date();
var flag6 = (today instanceof java.util.Date);
```

B2.4 Casting (as)

Casting is done using the `as` keyword, following the variable to be cast. Types can be tested using the `instanceof` operator.

```
var pseudoRnd:Integer =
    (java.langSystem.currentTimeMillis() as Integer) mod 1000;

var str:java.lang.Object = "A string";
var inst1 = (str instanceof String);
```

B3 Strings

Strings largely behave like they do in Java, but there are a few interesting extra pieces of functionality specific to JavaFX Script.

B3.1 String literals and embedded expressions

String literals may be written using either single or double quotes to enclose their content. Two consecutive strings in the source (with nothing but whitespace in between) are concatenated, even if separated by a new line.

```
var str1 = 'Single quotes';
var newline = "This string starts here, "
'and ends here!';
```

The same type of quote character must start and end a string. The quote not being used as a delimiter is free to be used inside the string. Backslash can be used to escape a quote.

```
println("UK authors prefer 'single quotes'");
println('US authors prefer "double quotes"');
println('I use "US" and \'UK\' quotes');
```

Embedded expressions may be run inside strings, using curly brace delimiters.

```
var rating = "cool";
var eval1 = "JavaFX is {rating}!";
var flag = true;
var eval2 =
    "JavaFX is {if(flag) "cool" else "uncool"}!";
```

B3.2 String formating

Embedded string expressions may also contain formatting, using the same syntax as Java's `java.util.Formatter` class.

```
import java.util.Calendar;
import java.util.Date;
def magic = -889275714;
println("{magic} in hex is {%08x magic}");
def cal:Calendar = Calendar.getInstance();
cal.set(1,3,4);
def joesBirthday:Date = cal.getTime();
println("Joe was born on a {%tA joesBirthday}");
```

B3.3 String localization

Strings can be localized using property files on the classpath. This permits our software to speak many different languages, and allows us to add new languages easily. The property filename should follow one of two formats:

```
<SCRIPT_NAME>_<LANG_CODE>.fxproperties
```

```
<SCRIPT_NAME>_<LANG_CODE>_<REGION_CODE>.fxproperties
```

In the above `<SCRIPT_NAME>` is the base name (no `.fx` extension) of the script to which the localization file applies, `<LANG_CODE>` is an ISO language code, and `<REGION_CODE>` is an ISO region code.

To use the localized strings in our programs we use a double # syntax, which has two variants, as follows:

```
def str1:String = ##"Trashcan";
def str2:String = ##[TRASH_KEY]"Trashcan";
```

The first example uses "Trashcan" as both the property search key, and the fall-back value. The second example uses "TRASH_KEY" as the property key, and "Trashcan" as the fall-back default.

B4 Durations

Durations are objects for storing and manipulating time. They have their own literal syntax, with a postfix `h` (hours), `m` (minutes), `s` (seconds) or `ms` (milliseconds) denoting the time units.

```
def mil = 25ms;
def sec = 30s;
def min = 15m;
def hrs = 2h;

def dur1 = 15m * 4;           // 1 hour
def dur2 = 0.5h * 2;          // 1 hour
def flag1 = (dur1 == dur2);   // True
def flag2 = (dur1 > 55m);     // True
def flag3 = (dur2 < 123ms);   // False
```

B5 Sequences: JavaFX arrays

JavaFX has no primitive arrays, as such, but what it has instead are single dimensional lists which can be declared and manipulated in numerous ways not supported by conventional Java arrays.

B5.1 Basic sequence declaration and access (sizeof)

A sequence may be declared with with its initial items between square brackets. A sequence inside a sequence is expanded in place. Two sequences are equal if contain the same quantity, type, value and order of items. The `sizeof` operator is used to determine the length of a sequence.

```
def seq1:String[] = [ "A" , "B" , "C" ];
def seq2:String[] = [ seq1 , "D" , "E" ];
def flag1 = (seq2 == ["A","B","C","D","E"]);
def size = sizeof seq1;
```

To reference a value we use the square bracket syntax. Referencing an index outside the current range returns a default value, rather than an exception.

```
def faceCards = [ "Jack" , "Queen" , "King" ];
var king = faceCards[2];
def ints = [10,11,12,13,14,15,16,17,18,19,20];
var oneInt = ints[3];
var outside = ints[-1];  // Returns zero
```

B5.2 Sequence creation using ranges ([..], step)

Sequences may be populated using a range syntax. An optional step may be supplied to determine the increment between items.

```
def seq = [ 1 .. 100 ];
def range1 = [0..100 step 5];
def range2 = [100..0 step -5];
def range3 = [0..100 step -5];
def range4 = [0.0 .. 1.0 step 0.25];
```

We can also include ranges inside larger declarations. Recall, a nested sequence is expanded in place.

```
def blackjackValues = [ [1..10] , 10,10,10 ];
```

B5.3 Sequence creation using slices ([..<])

We can create a sequence range using a slice of indexes from another sequence. Here the `..` syntax includes the end index in the range, while `..<` excludes the end index.

```
def source = [0 .. 100];
var slice1 = source[0 .. 10];    // 0 to 10
var slice2 = source[0 ..< 10];   // 0 to 9
var slice3 = source[95..];        // 95 to 100
var slice4 = source[95..<];       // 95 to 99 (exclude last item)
```

B5.4 Sequence creation using a predicate

A predicate can be used to take a conditional slice from an existing sequence, creating a new sequence.

```
def source = [0 .. 9];
var lowVals = source[n|n<5];
def people =
    ["Alan","Andy","Bob","Colin","Dave","Eddie"];
var peopleSubset =
    people[s | s.startsWith("A")].toString();
```

B5.5 Sequence manipulation (insert, delete, reverse)

Sequences can be manipulated by inserting and removing elements dynamically. We can even reverse a sequence.

```
var seq1 = [1..5];
insert 6 into seq1;
insert 0 before seq1[0];
insert 99 after seq1[2];

var seq2 = [[1..10],10];
delete seq2[0];
delete seq2[0..2];
delete 10 from seq2;  // Delete any 10's
delete seq2;  // Delete whole sequence

var seq3 = [1..10];
seq3 = reverse seq3;
```

B6 Binds

Binds define a relationship between a data source and a data consumer. When the source changes the bind performs a 'minimal re-calculation' using only those parts of the bind expression affected by any change.

B6.1 Binding to variables (bind)

Bound variables are created using the `bind` keyword. They are read only, so we use `def` instead of `var`.

```
var percentage:Integer;
def progress = bind "Progress: {percentage}% finished";
for(v in [0..100 step 20])
{   percentage = v;
    println(progress);
}

var thisYear = 2008;
def lastYear = bind thisYear-1;
def nextYear = bind thisYear+1;
// lastYear=2007, thisYear=2008, nextYear=2009
thisYear = 1996;
// lastYear=1995, thisYear=1995, nextYear=1996
```

It is possible to bind to a bound variable, as follows:

```
var flagA = true;
def flagB = bind not flagA;   // Always opposite of flagA
def flagC = bind not flagB;   // Always opposite of flagB, same as flagA
```

B6.2 Binding to a sequence

Binding to a sequence index is done so by way of its index.

```
var range = [1..5];
def ref = bind range[2];   // ref == 3
delete range[0];           // ref == 4
delete range;              // ref == 0
```

We can also bind against an entire sequence.

```
var seqSrc = [ 1..3 ];
def seqDst = bind for(x in seqSrc) { x*2; }   // seqDst == [2,4,6]
insert 10 into seqSrc;                        // seqDst == [2,4,6,20]
```

B6.3 Binding to code

The bound expression may contain code, including function calls.

```
var mode = false;
def modeStatus = bind if(mode) "On" else "Off";

var userID = "";
def realName = bind getName(userID);
function getName(id:String) : String
{   if(id.equals("adam")) { return "Adam Booth"; }
    else if(id.equals("dan")) { return "Dan Jones"; }
    else { return "Unknown"; }
}
```

B6.4 Bidirectional binds (with inverse)

For simple binds, which mirror another variable directly, its possible to create a two-way relationship, such that changing one variable changes the other. Because this type of bind is assignable, we use `var` instead of `def`.

```
var dictionary = "US";
var thesaurus = bind dictionary with inverse;
// dictionary=”US”, thesaurus=”US”
thesaurus = "UK";
// dictionary=”UK”, thesaurus=”UK”
thesaurus = "FR";
// dictionary=”FR”, thesaurus=”FR”
```

B6.5 Bound functions (bound)

External variables functions may use are not included in the bind. To change this, mark the function as `bound`, and its dependencies will trigger bind updates correctly.

```
var ratio = 5;
var posX = 5;
def coord = bind "{scale(posX)}";  // Binds posX and ratio
bound function scale(v:Integer) : Integer
{   return v*ratio;
}
```

B7 Quoted identifiers

Quoted identifiers mark a part of the source code as an identifier, which otherwise might have confused the compiler. They are handy for referencing Java method names which clash with JavaFX Script reserved words, for example.

```
var <<var>> = "A string variable called var";
```

B8 Packages (package, import)

Packages relate portions of our code together, into a group. They work the same way as Java packages. Importing allows the code to avoid using cumbersome fully qualified class names. As in Java, an asterisks can be used at the end of an `import` statement instead of a class name, to include all the classes from the stated package without having to list them individually.

```
import java.util.Date;
var date1:Date = Date {};
```

Adding a class into a package is as follows:

```
package jfxia.chapter3;
public class Date
{   override function toString() : String
    {   "This is our date class";
    }
};
```

Using both Java's `Date` class and the one above looks like this:

```
import java.lang.System;
var date1 = java.util.Date {};
var date2 = jfxia.chapter3.Date {};
```

B9 Developing classes

JavaFX Script's support for OO is pretty rich, including multiple inheritance. Classes referenced outside of a source file must live in an individual file named after said class. Unlike with Java, not all code in JavaFX Script has to live inside a class – so called *script level* code is bundled up and run as if it is in a Java `main()` method.

B9.1 Scripts

Variables and functions can live at script level; that is, outside of any class. When located in the script context, they behave like (and are accessed like) Java static class members.

```
// This code lives in a file called "Examples2.fx"
package jfxia.chapter3;

def scriptVar:Integer = 99;
function scriptFunc() : Integer
{   def localVar:Integer = -1;
    return localVar;
}

println
(   "Examples2.scriptVar = {Examples2.scriptVar}\n"
    "Examples2.scriptFunc() = {Examples2.scriptFunc()}\n"
    "scriptVar = {scriptVar}\n"
    "scriptFunc() = {scriptFunc()}\n"
);
```

Any loose code which lives in at the script level (code not part of a script function) can be executed as an application bootstrap.

B9.2 Class definition (class, def ,var, function, this)

Variables in classes are creared with the familiar `var` and `def` keywords, while behavior is implemented by way of a `function`. The keyword `this` can be used to refer to the current object, although it's use is optional.

```
import javafx.lang.Duration;
class Track
{   var title:String;
    var artist:String;
    var time:Duration;
    function equals(t:Track) : Boolean
    {   return (t.title.equals(this.title) and
            t.artist.equals(this.artist));
    }
    override function toString() : String
    {   return '"{title}" by "{artist}" : '
            '{time.toMinutes() as Integer}m '
            '{time.toSeconds() mod 60 as Integer}s';
    }
}
var song:Track = Track
{   title: "Special"
    artist: "Garbage"
    time: 220s
};
```

B9.3 Object declaration (init, postinit, isInitialized(), new)

JavaFX Script objects have no constructors, preferring its trademark declarative syntax instead. The `init` block of code is called once all class variables have been assigned, and once finished the object is considered initialized. The `postinit` block is then called.

The built in function `isInitialized()` can be used to find out if a variable was assigned a value during object initialization.

```
def UNKNOWN_DRIVE:Integer = 0;
def WARP_DRIVE:Integer = 1;
class SpaceShip
{   var name:String;
    var crew:Integer;
    var canTimeTravel:Boolean;
    var drive:Integer = SpaceShip.UNKNOWN_DRIVE;
    init
    {   println("Building: {name}");
        if(not isInitialized(crew))
            println("  Warning: no crew!");
    }
    postinit
    {   if(drive==WARP_DRIVE)
            println("  Engaging warp drive");
    }
}

def ship1 = SpaceShip
{   name:"Starship Enterprise"
    crew:400
    drive:SpaceShip.WARP_DRIVE
    canTimeTravel:false
};
```

There may be times when we are forced to instantiate a Java object using a specific constructor to make it function the way we desire. The `new` syntax allows us to do just that. It can be used on any class, including JavaFX Script classes, although it is intended to be used only with Java classes, when the declarative syntax isn't sufficient.

```
def ship2 = new SpaceShip();
ship2.name="The Liberator";
ship2.crew=7;
ship2.canTimeTravel=false;
```

B9.4 Class inheritance (abstract, extends, override)

JavaFX Script supports inheritance, using the `extends` keyword. As in Java, functions are virtual. The `abstract` keyword can be used to prevent a class from being instantiated directly. The `override` keyword is needed on any instance function or variable which overrides a parent class; overridden variables are used to change initial values.

The following code should live in a file called `Animal.fx` :

```
import java.util.Date;

abstract class Animal
{   var life:Integer = 0;
    var birthDate:Date;
```

```
        function born() : Void
        {   this.birthDate = Date{};
        }
        function getName() : String
        {   "Animal"
        }
        override function toString() : String
        {   "{this.getName()}  Life: {life}  "+
                "Bday: {%te birthDate} {%tb birthDate}";
        }
    }
```

The following class is a subclass of the above `Animal` class:

```
    class Mammal extends Animal
    {   override function getName() : String
        {   "Mammal"
        }
        function giveBirth() : Mammal
        {   var m  = Mammal { life:100 };
            m.born();
            return m;
        }
    }
```

JavaFX Script permits multiple inheritance, whereby a class can inherit several JavaFX classes in one go. The `extends` syntax supports a comma separated list of classes.

```
    abstract class Motor
    {   var distance:Integer;
        function move(dir:Integer,dist:Integer) : Void
        {   // Move in direction
            distance+=dist;
        }
    }
    abstract class Weapon
    {   var bullets:Integer;
        function fire(dir:Integer) : Void
        {   if(bullets>0)
            {   // Fire weapon
                bullets--;
            }
        }
    }
    class Robot extends Motor,Weapon
    {   var name:String;
        function turn() : Void
        {   var dir = 0;
            // Calculate direction to move
            move(dir,1);
            fire(dir);
        }
    }
```

In order to co-operate with Java, JavaFX Script makes a distinction between classes it refers to as being *compound* and those it refers to as being *plain*. A *plain class* is any class which extends, directly or indirectly, a regular Java class. A *compound class* is any class entirely constructed from within JavaFX.

- All JavaFX classes are compound by default.

- A JavaFX class may directly inherit any number of compound classes, but only one plain class.
- Any JavaFX class which inherits a plain class, becomes a plain class.

B9.5 Multiple inheritance and the diamond problem

The diamond problem occurs in any Object Oriented language which supports multiple inheritance. It involves a class inheriting two classes which share the same function, thereby creating a name clash.

There's little official documentation for JavaFX Script v1.0 to explain how it tackles this problem, but the following experimental source code demonstrates the problem, what works, and what does not. The comment lines in bold are compiler/stdout output for the previous line.

```
// Filename: Diamond.fx
class Base
{
}
class Parent1 extends Base
{   override function toString() : String
    {   "This is Parent1";
    }
}
class Parent2 extends Base
{   override function toString() : String
    {   "This is Parent2";
    }
}
class Child extends Parent1, Parent2
{   public function printContents() : Void
    {   println("{Parent1.toString()}");
        println("{Parent2.toString()}");
    }
}

def a = Child{}

// Test 1
a.printContents();
// This is Parent 1
// This is Parent 2

// Test 2
// println("{a.Parent1.toString()}");
// Causes a compiler error...
// Cannot find symbol: variable Parent1

// Test 3
// println("{a.toString()}");
// Causes a compiler error...
// Reference to toString is ambiguous, both
//   method toString() in Diamond.Parent2 and
//   method toString() in Diamond.Parent1 match

/ Test 4
```

```
def b:Base = a as Base;
println("{b.toString()}");
// This is Parent2

// Test 5
def c:Parent1 = a as Parent1;
println("{c.toString()}");
// This is Parent2
```

For this example we can see the following:

- Test 1 shows it is possible to reference a specific disputed function by qualifying the reference with the name of the host parent class.
- Test 2, however, shows prefixing the class name like this causes a compiler error when outside the class itself.
- Test 3 shows a direct call to a disputed function also results in a compiler error.
- But test 4 demonstrates a direct call is possible when the object is cast to the base type. In this case, it's the function from the later class on the `extends` list which is called.
- And test 5 indicates the later class on the `extends` list is always used, even when the object is explicitly cast to an earlier class on the list.

B9.6 Function types

Functions in JavaFX Script are *first class objects*, meaning we can have variables of function type, and even pass a function into another function as a parameter.

```
var func : function(:String):Boolean;
func = testFunc;
function testFunc(s:String):Boolean
{   return (s.equalsIgnoreCase("true"));
}
```

The variable `func` above is of type `function(:String):Boolean`, which represents the signature of functions which may be assigned to it. Functions may be passed as parameters to other functions using a similar syntax.

```
function manip(s:String , f:function(:String):String) : Void
{   println("{s} = "+f(s));
}
function m1(s:String) : String
{   s.toLowerCase();
}
function m2(s:String) : String
{   s.substring(0,4);
}
manip("JavaFX" , m1);
manip("JavaFX" , m2);
```

B9.7 Anonymous functions

Anonymous functions allows lightweight, single-use, nameless, functions to be created within a declarative context.

```
import java.io.File;
```

```
class FileSystemWalker
{   var root:String;
    var extension:String;
    var action:function(:File):Void;

    function go() { walk(new File(root)); }
    function walk(dir:File) : Void
    {   var files:File[] = dir.listFiles();
        for(f:File in files)
        {   if(f.isDirectory())
            {   walk(f);
            }
            else if(f.getName().endsWith(extension))
            {   action(f);
            }
        }
    }
}

var walker = FileSystemWalker
{   root: FX.getArguments()[0];
    extension: ".png";
    action: function(f:File)
    {   println("Found {f.getName()}");
    }
};
walker.go();
```

In the above example an anonymous function, accepting a `File` as a parameter, is assigned to the `action` variable of a `FileSystemWalker` class instance. Anonymous functions similar to this are used extensively for event handling in JavaFX Script, instead of Java's class-heavy listener model.

B9.8 Access modifiers (package, protected, public, public-read, public-init)

Using access modifiers we can restrict access to our script or instance variables and functions. Access modifiers perform the same role in JavaFX Script as they do in Java, but their implementation differs from Java.

Table B1 Basic access modifiers

Modifier keyword	Visibility effect
(default)	Visible only within the enclosing script. This default mode (with no associated keyword) is the least visible of all access modes.
`package`	Visible within the enclosing script, and any script or class within the same package.

Modifier keyword	Visibility effect
`protected`	Visible within the enclosing script, any script or class within the same package, and subclasses from other packages. This modifier only works with class members, not script members or the class itself.
`public`	Visible to anyone, anywhere.

Tables B2 Additive access modifiers

Modifier keyword	Visibility effect
`public-read`	Adds public read access to the basic mode.
`public-init`	Adds public read access and object literal write access to the basic mode.

The comments above each class, variable and function, in the following code explains their visibility.

```
// Instantiate: anywhere
public class AccessTest
{   // Class and script only
    var sDefault:String;
    // Class, script and package
    package var sPackage:String;
    // Class, script, package, and any subclass
    protected var sProtected:String;
    // Everywhere
    public var sPublic:String;

    // Write: class and script only / Read: anywhere
    public-read var sPublicReadDefault:String;
    // Write: class, script and package / Read: anywhere
    public-read package var sPublicReadPackage:String;
    // Write: class, script, package and subclass – plus anywhere
    //   when creating object literals / Read: anywhere
    public-init protected var sPublicInitProtected:String;
}
```

B10 Conditions

JavaFX Script's conditions look and behave in a not too dissimilar fashion to the conditions of other languages, Java included, however JavaFX Scripts expression based syntax affords some interesting twists.

B10.1 Basic conditions (if, else)

The if/else syntax is the same as Java's. JavaFX Script has no keyword for else/if – instead we should use an `if` construct directly after an `else`.

```
ivar someValue = 99;
```

```
if(someValue==99)
{   println("Equals 99");
}
if(someValue >= 100)
{   println("100 or over");
}
else
{   println("Less than 100");
}

if(someValue < 0)
{   println("Negative");
}
else if(someValue > 0)
{   println("Positive");
}
else
{   println("Zero");
}
```

Because JavaFX's conditions are expressions they give out a result, and as such can be used on the right hand side of an assignment, or as part of a bind, or any other situation in which a result is expected.

```
var negValue = -1;
var sign = if(negValue < 0) { "Negative"; }
    else if(negValue > 0) { "Positive"; }
    else { "Zero"; }
```

B10.2 Ternary expressions, and beyond

As all conditions are also expressions, JavaFX script requires no explicit syntax for ternary expressions.

```
var asHex = true;
java.lang.System.out.printf
(   if(asHex) "Hex:%04x%n" else "Dec:%d%n" ,
    12345
);
```

Using the expression language syntax it is possible to go beyond the standard true/false embedded condition.

```
var mode = 2;
println
(   if(mode==0) "Yellow alert"
    else if(mode==1) "Orange alert"
    else if(mode==2) "Mauve alert"
    else "Red alert"
);
```

B11 Loops

Loop allow us to repeatedly execute a given section of code until a given condition is met. In JavaFX Script `for` loops are tied firmly to sequences, and work as expressions just like conditions. A more traditional type of loop is the `while` loop, which runs a block of code until a condition is met.

B11.1 Basic sequence loops (for)

For loops work like the for/each loop of other languages.

```
for(a in [1..3])
{   for(b in [1..3])
    {   println("{a} x {b} = {a*b}");
    }
}
```

For loops are expressions, returning a sequence.

```
var cards =
    for(str in ["A",[2..10],"J","Q","K"])
        str.toString();
```

In the above, each pass through the loop an element is plucked from the source sequence, converted into a string, and added to the destination sequence, `cards`.

B11.2 Rolling nested loops into one expression

The loop syntax gives us an easy way to create loops within loops, allowing us, for example, to drill down to access sequences within sequences.

```
import java.lang.System;
class SoccerTeam
{   var name: String;
    var players: String[];
}
var fiveAsideLeague:SoccerTeam[] =
[   SoccerTeam
    {   name: "Java United"
        players: [ "Smith","Jones","Brown",
            "Johnson","Schwartz" ]
    } ,
    SoccerTeam
    {   name: ".Net Rovers"
        players: [ "Davis","Taylor","Booth",
            "Williams","Ballmer" ]
    }
];
for(t in fiveAsideLeague, p in t.players)
{   println("{t.name}: {p}");
}
```

B11.3 Controlling flow within for loops (break, continue)

JavaFX Script supports both the `continue` and `break` functionality in its for loops, to skip to the next iteration, or terminate the loop immediately. Loop labels are not supported.

```
var total=0;
for(i in [0..50])
{   if(i<5) { continue; }
    else if (i>10) { break; }
    total+=i;
}
```

B11.4 Filtering for expressions (where)

Filters selectively pull out only the elements of the source sequence we want.

```
var divisibleBy7 =
    for(i in [0..50] where (i mod 7)==0) i;
```

The result of the above is a sequence who's contents are only those numbers from the source evenly divisible by seven.

B11.5 While loops (while, break, continue)

JavaFX Script supports `while` loops, in a similar fashion to other languages. As with `for` loops, the `break` keyword can used to exit a loop prematurely (labels are not supported) and `continue` can be used to skip to the next iteration.

```
var i=0;
var total=0;
while(i<10)
{   total+=i;
    i++;
}
i=0;
total=0;
while(i<50)
{   if(i<5) { i++; continue; }
    else if (i>10) { break; }
    total+=i;
    i++;
}
```

B12 Triggers

Triggers allow us to assign some code to run when a given variable is modified. Triggers can be used with either regular variables, or sequences.

B12.1 Single value triggers (on replace)

The trigger syntax is used at the end of a variable declaration, with the keywords `on replace` and two variables, one for the current value and one for the incoming value. In the below example, `previous` always holds the last value of its companion, `current`.

```
class TestTrigger
{   var current = 99 on replace oldVal = newVal
    {   previous = oldVal;
    };
    var previous = 0;
}
```

It is permitted to miss off the new value and just use the variable name itself, or to miss off both values.

```
var a = 99 on replace oldVal
{   println("Old={oldVal}  New={a}");
}
var b = 99 on replace
{   println("New={b}");
}
```

B12.2 Sequence triggers (on replace [..])

We can also assign a trigger to a sequence. To do this we need to also tap into not only the existing and replacement values, but the range of the sequence which is being affected. The

old and new values refer to sequences, and a range-like syntax is used to give the index span of the elements affected.

```
var seq1 = [1..3]
    on replace oldVal[lo..hi] = newVal
    {   println
        (   "Changing [{lo}..{hi}] from "+
            "{oldVal.toString()} to "+
            "{newVal.toString()}"
        );
    };
```

For an insert the *low index* is the insert point, the *high index* is the low index minus one, the *old sequence* is empty, and the *new sequence* contains the values being added. For a delete the *low index* and the *high index* define the range being removed, the *old sequence* is the sequence as it currently is, and the *new sequence* is empty. For a change (including a reverse) the *low index* and *high index* define the range, the *old sequence* is the content as it currently is, and the *new sequence* is the content after the change is applied.

B13 Exceptions (try, catch, any, finally)

Exceptions give us a way to assign a block of code to be run when a problem occurs, or to signal a problem within our own code to outside code which may be using our API. Catch blocks can trap exceptions of a particular type, or the any keyword can be used to create a *catch-all* default exception handler. Finally blocks are always run when the try block exits, whether cleanly or as the result of an exception. As in Java, the try/finally construct may be used on their own, without any exception handling blocks.

```
import java.lang.NullPointerException;
import java.io.IOException;

var key = 0;
try
{   println(doSomething());
}
catch(ex:IOException)
{   println("ERROR reading data {ex}")
}
catch(any)
{   println("ERROR unknown fault");
}
finally
{   println("This always runs");
}

function doSomething() : String
{   if(key==1)
    {   throw new IOException("Data corrupt");
    }
    else if(key==2)
    {   throw new NullPointerException();
    }
    "No problems!";
}
```

B14 Keywords

The following table lists JavaFX Script keywords and reserved words. Implemented keywords are shown as regular text, reserved (but unused) words are shown underlined.

abstract	after	and	as	assert
at	attribute			
before	bind	bound	break	
catch	class	continue		
def	delete			
else	exclusive	extends		
false	finally	first	for	from
function				
if	import	indexof	in	init
insert	instanceof	into	inverse	
last	lazy			
maxin	mod			
new	not	null		
on	or	override		
package	postinit	private	protected	public-init
public	public-read			
replace	return	reverse		
sizeof	static	step	super	
then	this	throw	trigger	true
try	tween	typeof		
var				

```
where          while          with
```

B15 Operator Precedence

The following table lists operators, grouped by precedence (the priority order in which they take effect.)

Pri.	Operator	Evaluation mode	Description
1	`function()`	Class	Function call
1	`()`		Bracketed expression
1	`new`	Class	Object instantiation
1	*Object literal*	Class	Object instantiation and initialization
2	`++ (suffixed)`	Right to left	Post increment assign
2	`-- (suffixed)`	Right to left	Post decrement assign
3	`++ (prefixed)`	Right to left	Pre-increment assign
3	`-- (prefixed)`	Right to left	Pre-decrement assign
3	`not`	Boolean	Logical negation
3	`sizeof`	Sequence	Sequence length
3	`reverse`	Sequence	Sequence reverse
3	`indexof`	Sequence	Element index in sequence
3	`=>`		Tween
4	`*`	Left to right	Multiplication
4	`/`	Left to right	Division
4	`mod`	Left to right	Remainder
5	`+`	Left to right	Addition

Pri.	Operator	Evaluation mode	Description
5	-	Left to right	Subtraction
6	==	Left to right	Equality
6	!=	Left to right	Inequality
6	<	Left to right	Less than
6	<=	Left to right	Less than or equal to
6	>	Left to right	Greater than
6	>=	Left to right	Greater than or equal to
7	instanceof	Class	Type check
7	as	Class	Type cast
8	or	Right to left	Logical OR
9	and	Right to left	Logical AND
10	+=		Add with assign
10	-=		Subtract with assign
10	*=		Multiple with assign
10	/=		Divide with assign
10	%-		Remainder with assign
11	=		Assign

B16 Pseudo variables

JavaFX Script is host to a handful of handy pre-defined global variables, for accessing environment and script/class details.

Variable	Purpose

Variable	Purpose
__DIR__	Returns the location of the current class file, as a URL. A directory if the class is a regular bytecode file, or a JAR file if the class is inside a Java archive.
__FILE__	Rerturns the fill filename and path of the current class file, as a URL.

Appendix C

Not familiar with Java?

This section is a collection of brief articles which back up the material in the language tutorial chapters, and elsewhere. Many of this book's readers will have arrived at JavaFX from Java, but not all! JavaFX is deliberately designed to have a broad appeal beyond just the regular Java desktop programmers. The problem was this: how to supply the necessary background knowledge about the Java platform to the latter, without boring the former? This appendix is the solution.

Each section in this appendix is a small essay dealing, in reasonable detail, with those nuanced workings of the Java platform which apply to JavaFX, or other background material.

C1 Static types versus dynamic types

Although JavaFX Script started life as a scripting language it does not adopt the loose laissez-faire attitude to variable types found in other scripting languages. If you came here from a design background (perhaps with some basic scripting experience thanks to older versions of JavaScript or ActionScript) you may not be familiar with how a statically typed language differs from the dynamically typed code you're used to.

```
someVariable = "123"
someVariable = someVariable+1
```

The above code fragment (in no particular language) demonstrates the difference between JFX's static typing and the dynamic typing of other scripting languages. The variable `someVariable` is being loaded with the string "123", then an arithmetic operation is being performed on its contents – what should the result of this operation be?

One answer might be to throw a runtime error, as arithmetic cannot be performed on text. Another answer might be to silently normalize the two operands, either by converting the string to a number (resulting in the value 124) or the value to a character string (resulting in the string "1231".) Or the operation could simply be flagged as invalid before the code runs – a string is a string, a value is a value, and never the twain shall meet

(except under strict predefined conditions, where runtime consequences can always be anticipated.)

JavaFX Script, following Java's lead, uses the static typed solution. Variables are clearly delineated as to their content type, and by inference what operations are valid on them. This approach can sometimes add a little coding overhead, for example explicit operations are needed to convert user input (typically character data) into value types, but the plus side is it might just spare us a few embarrassing crashes when the company CEO stupidly types "Forty Nine" into our application's age field during a live demo.

C2 Casts

In statically typed languages (see C1 above) the type of a variable is important, but there are times when data doesn't arrive in the form you want it to. Casting is a way of asking JavaFX Script to translate one data type into another compatible type. Most programming languages insist on casts when there's risk of a potential loss of information, for example if a 64 bit value is stored in a 32 bit variable. Casting generally isn't required to go the other way, because the operation is guaranteed to be safe from data loss.

```
import java.lang.System;
var pseudoRnd:Integer =
    (System.currentTimeMillis() as Integer) mod 1000;
```

In the example above we want the take just the milliseconds part of the current time, to use as a very rough pseudo random number between 0 and 999. The Java API method `System.currentTimeMillis()` returns the current time as the total number of milliseconds since the Unix/POSIX epoch (midnight on 1st January 1970) which returns a 64 bit number. As it only takes a regular 32 bit integer to store a value between 0 and 999, we cast the result to an JFX `Integer`, before taking only the lower three decimal digits.

As you'll have witnessed above, JavaFX uses the keyword `as` to perform casts, with type to which the data it to be translated at the tail end of the expression. This contrasts with the syntax of Java, C, C++ and others, where the type prefixes the data surrounded in parenthesis.

As previously hinted, we can often get away without using a cast, but sometimes they are necessary, and the compiler will insist we use one. The most common usages are when there's risk of potential data loss (the aforementioned 64 to 32 bit example), when we need to explicitly spell out what type an object is because the information isn't available to the compiler (for example, when an object has been stored as a super type) or our code is ambiguous (for example, the parameters of a method call could match more than one overloaded alternative.)

C3 Packages

Packages allow us to relate portions of our code together, into a group. There are several reasons why this might be handy, including...

1 Convenience: just as grouping files into directories imposes order on our data, so grouping code into packages can impose order within our software. Anything which

allows us to organize our code so we can better manage it is useful when writing non-trivial applications.

2 Integrity: it's possible to allow functions and variables to be visible to other members of a package, yet deny access to non-members (see *access modifiers*, later.) This permits creation of functionality which spans several classes, without exposing implementation details to third party developers who may be using our API. It means we can write sophisticated software, yet still lock other programmers out of parts we wish to control ourselves.

3 Flexibility: we can have two classes which share the same name, providing they live in different packages. This means we can use third party APIs without fear of class naming clashes.

Throughout the source in this book you'll see the line `import java.lang.System;` crop up frequently at the start of many of the pieces of code. This line is necessary to allow us to refer to the `System` object (as in the `System.out.println()` calls) without using its *fully qualified class name*.

Let's take a look at an example:

```
import java.util.Date;                                          A
                                                                A
var date1:Date = Date {};                                       A
var date2:java.util.Date = java.util.Date {};                   A
```

A "Date" lives in the package "java.util"

We see two different ways of creating a `Date` object above – the first makes use of the `import` statement at the start of the code (and would fail without it), while the second does not. Importing classes means we, the programmers, can avoid getting repetitive strain injuries typing in those long, and frankly quite ugly, package prefixes each time we wish to refer to a given class. The `import` statement is a *heads-up* to the compiler as to which classes are likely to appear in the current source code file. The compiler will then silently add the missing package prefix to any class reference which does not have one.

As in Java, an asterisks can be used at the end of an `import` statement instead of a class name, to include all the classes from the stated package without having to list them individually.

But how do we create our own packages, and what happens if we need to work with two classes which share an identical name, but live in different packages? The next example will deal with both these issues. But first we need to create a demonstration class – we haven't discussed creating our own classes yet (but we will shortly) so the next bit of code is a small preview of what's to come.

```
package jfxia.chapter3;                                         A
                                                                A
class Date                                                      A
{   function toString() : String                                A
    {   "This is our date class";                               A
    }                                                           A
};                                                              A
```

A Should go in the file "Date.fx"

Above we have the definition for a class called `Date` (in the file `Date.fx`, so the compiler can find it) which does nothing more than print a message when its `toString()` function is called. The `package` statement at the head of a source file places the code into `jfxia.chapter3`, which is the package we want it to live in. Now we need some more code to test it:

```
import java.lang.System;

var date1 = java.util.Date {};
var date2 = jfxia.chapter3.Date {};
System.out.println(date1.toString());
System.out.println(date2.toString());
```

```
Mon Aug 04 18:33:03 BST 2008
This is our date class
```

We didn't bother to import our package! Why?

Well in this example we're using both our own `Date` class, and the one in the Java API package `java.util` – which is a naming conflict: we now have two classes with the same name. If we neglect to provide their *fully qualified class name* (the class name including its package prefix) how can the compiler tell which class we are referring to?

The answer is: it can't! We need to provide the fully qualified name of each class to avoid ambiguity. This makes including imports for the classes at the start of our code rather academic – although the import wouldn't throw a compilation error, we wouldn't be able to take advantage of it by using abbreviated names for either `Date` class, not so long as we have a clash of names in our source code file.

PACKAGES ARE NOT A HEIRARCHY

Despite the misleading impression their names often suggest, Java and JFX packages are not arranged in a hierarchy. The package `java.awt.event` is actually a sibling of `java.awt`, not a child. If you import all the classes from the latter, you do not automatically get the former.

By convention all of the classes in a package are bundled inside a single library Jar file – it's possible to split a package over multiple Jars, but the practice is rarely necessary – with a single Jar sometimes containing multiple related packages.

C4 Object Orientation

Classes are an integral part of Object Orientation, encapsulating state and behavior for each component in a larger system, thereby allowing us to express our software in terms of the structures and relationships which link its autonomous component parts. Object Orientation has become an incredibly popular way of constructing software in recent years: both Java, and its underlying Java Virtual Machine environment, are heavily object centric. But what is Object Orientation?

At the sharp end of Object Oriented software everything tends to boil down to types. What type is an object? For example, is it a plane, a train, or an automobile? Of course, all

three are types of vehicle, and share common properties and functionality. They all move, and therefore have a speedometer, an odometer (mileage) and consume power. They all carry passengers as well. They all need some form of engine to drive them forward, and a braking system to slow them down. But, of course, they also have a lot of differences. Trains cannot arbitrarily turn left or right, because they are bound by the constraints of a track – or at least they shouldn't be able to under normal operating conditions. Cars cannot fly through the air like a plane (again, under normal operating conditions!) but they can reverse, which is something a plane in flight cannot do (or can it? See later.)

We build up Object Oriented software by modeling these relationships. Classes are the nodes we link together to create such models. If we were building a transport simulator we might start with a vehicle class which contained all the data and functionality we know is common to all vehicles in our system. The odometer, for example, could be included in this top level class, because all vehicles have a mileage. We could also define a few functions, perhaps one to speed up and one to slow down, because both of these are common to all vehicles. Yet each vehicle accelerates and decelerates in a different way, so these functions can only be stubs which will be populated by the specific vehicles themselves.

We could then start to define new classes of vehicle based on our top level class. We might define a plane class, which adds an altitude attribute. We might also define an automobile class, which adds turn left and turn right functions. And so on... The process of creating a more specific class in terms of a more general one is known as *subclassing*. We say that (in terms of our simulator) a plane is a subclass of vehicle – in other words it is a type of vehicle, and can be treated in our simulation either specifically as a plane, or generically as a vehicle.

When the plane subclasses vehicle it can fill out the stub functions for accelerating and decelerating with some code which makes sense to how planes work. When it does so we say the plane has *overridden* those functions – in other words it has replaced the ones in vehicle with its own more specialized implementations. Later on we might define a specific class for a Harrier Jump Jet, which has very different movement capabilities to other planes (it can hover on the spot), and so may need to override the plane's accelerate and decelerate functions again, with even more Harrier-specialized versions.

An object on our simulation may be created as a type of Harrier Jump Jet, which in turn is a type of plane, which in turn is a type of vehicle. Because we know the Harrier inherited all the functionality of the plane (even if it did override some of it with its own implementation), and by proxy inherited all the functionality of vehicle (again, even if it replaced some of it), the Harrier can be treated as either a Harrier, a plane, or a vehicle. This means we can add the Harrier object to a vehicle array containing stream trains, Ford Model T's and Apollo space capsules – all of which will ultimately inherit from our top level vehicle class in the simulation.

When we treat the Harrier as a vehicle we can only reference those attributes and functions which are available to vehicles – so checking the odometer is fine because it is part of the vehicle class, but checking the altimeter will result in a compilation error. The

altimeter is specific to the place class, and its subclasses. To use it we need to cast the generic vehicle object back into a more-specific plane or Harrier Jump Jet object. This ability to treat an object by way of its super types (its parents in the class hierarchy) is known as *polymorphism*.

C5 Access modifiers

Access modifiers allow us to control the visibility of parts of our class.

Consider the following scenario: as part of a larger system we constructed a class which dealt with dates, but for some reason we only bothered to record the last two digits of the year. So 2005 is stored as only 05. This isn't a problem, because the class supports a `getYear()` function which adds on 2000 before it returns a result. Then our boss comes to see us and explains the system is being expanded to deal with data from as far back as 1995 – time to change our class to store dates as four digits. But as soon as we publish the change a fellow programmer from another part of the team complains we're making his code break! Extensively throughout his code he was reading the year directly, not bothering with our `getYear()` function, and so what we assumed would be a minor change is now a major headache.

What we need is some way to lock other programmers out of the implementation detail of our code, to effectively mark parts of the code as "no go" areas, and force everyone to use a class the way we intended it to be used. Access modifiers provide just such a mechanism.

- At one extreme end of the visibility scale we have public. Any function or attribute defined with a `public` prefix (access modifier) is visible to the whole world. Anyone can call said function, and anyone read or write said attribute.
- The next level up is protected, which affords some degree of privacy. Only code from the same package, or subclasses, can see functions or attributes prefixed as `protected`.
- The next level is the default level, sometimes called "package-private", which requires no keyword prefix. This is the same as protected, except subclasses are not included – only code from the same package can view attributes and functions with default visibility.
- The final, and tightest, level of visibility is private. Any attribute or function prefixed with `private` can only be seen by code within the same class – including free roaming code which lives outside the class body itself, but in the same source file.

If we'd used access modifiers in the example we began with, our fellow programmer wouldn't have been able to directly access the details of how we recorded the year in our class. Plus we'd feel confident in fixing bugs and upgrading the class internals, because we can be sure which parts of our code others can see, and which parts are under our total control.

Appendix D

JavaFX and the Java platform

JavaFX is not Java, but it rests within a sea of tools and technologies designed to support Java. As such it shares the unusual dualistic characteristic of being *of* Java (the platform), but not Java (the language)!

Given JavaFX's intentions it's reasonable to assume a minority of readers may have been drawn to it (and thereby this book) without first having come through Java; they would no doubt benefit from a little background. While Java-savvy readers will surely be keen to hear how the new platform and the old cooperate. So for young pups and old dogs alike, this appendix is a loose collection of small background articles introducing Java and exploring how JavaFX fits into the existing Java environment.

D1 How not to go native

The Java is a software platform which seeks to fulfill the mantra "write once, run anywhere". Software is compiled to bytecode files in the form of machine code instructions runnable on a virtual machine called a JVM (Java Virtual Machine.) The virtual machine provides a layer of abstraction, allowing the program to be run on many devices without needing to be specifically compiled to the machine code of the underlying hardware.

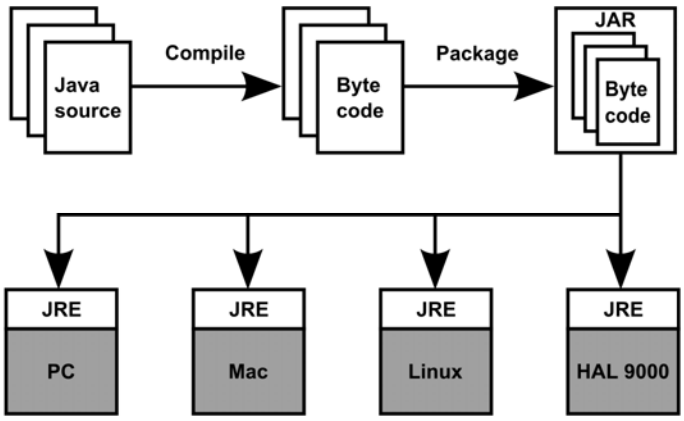


Figure D1 From compile-time to run-time: the life cycle of a typical Java application.

Once the source code files are compiled into bytecode class files they can be bundled into a single archive known as a JAR (Java ARchive) for easy distribution. JARs can contain runnable applications, or support libraries. There are numerous ways of getting the software onto the end-user computer, from the traditional (ship it on a CD-ROM) to the modern (embed it in a web page.)

JavaFX Script compiles to bytecode files, just like Java, although due to the way the language works each pure JavaFX Script class is translated into both a class and an interface. JavaFX Script can also access classes in Java package, both the standard API packages and any third party packages which are on the classpath when the JavaFX software is run.

D2 Java SE/ME/EE and JDK/JRE: three editions and two audiences

For programmers unfamiliar with Java, it should be noted there are three main Java markets: *Standard Edition* targets regular desktop users, *Micro Edition* is an extension to the Standard Edition adding tools and libraries for small devices like cell phones, and *Enterprise Edition* enhances the Standard Edition for writing web applications and web services.

JavaFX targets both the Standard and Micro editions. While applets are a web technology, they run on the client, so they're part of the Standard not the Enterprise Edition. The initial release of JavaFX, at the end of 2008, focuses on only the desktop (Standard Edition), with the mobile platform due to be covered in 2009. At the time of writing (early 2009) it is unclear how JavaFX Mobile and Java ME will relate. The mobile preview shipped with JavaFX 1.0 seems to be an alternative to JavaME on phones, rather than an extension. The BD-J (Blu-ray Disc Java) platform, likely to be targeted by JFX when it comes to TV

devices in 2009, is another variant of JavaME. As yet we do not know how JavaFX for TVs will relate to BD-J; will it compliment the existing JavaME based environment, or replace it?

Moving on, there exist two basic audiences for Java: the regular user who merely wants to run the software, and the programmer who wants to create the software. These are personified by the *Java Runtime Environment*, and the *Java Development Kit*. The former has the tools necessary to run a Java program and is often shipped pre-installed on desktop computers, while the latter bolsters the JRE with extra tools to create and debug Java software.

JavaFX has its own compiler which stands in for the regular Java compiler when writing JavaFX Script code, and its own software libraries more attune to its focus on graphics and animation.

D3 Release versions: a rose by any other name

One source of confusion for novice Java programmers is release names and versions. Since 1995 Java has been through many revisions. In the beginning it was simple; the first commonly used version of Java was 1.02, which was succeeded by 1.1 shortly thereafter. Someone then apparently decided 1.2 didn't sound important enough, so 'Java' acquired the nom-de-plume 'Java2'. Thus we got *Java2 Standard Edition version 1.2*, more commonly written as *J2SE v1.2*. This schizophrenic naming convention continued until version 1.5, when it was replaced by a new schizophrenic naming convention; 'Java2' reverted to 'Java' once more, and 1.5 became 5.0, however due to technical issues the 1.5 label continued to be used in some places.

(It is left as an exercise for the reader to guess what medication the people who came up with the above may have been on.)

Java SE 6 Update 10 (or later) is the version of the Java platform currently recommended for writing and running JavaFX software. The tenth update overhauls the applet plugin mechanics and provides a much smoother way to get Java onto the end user's machine – both of which are greatly beneficial to JFX. At the time of writing Update 10 is in early access beta only; by the time you read this book it should have made it to a full release.