3 JavaFX Script
CODE AND STRUCTURE

This chapter covers

- Writing, inheriting, and using classes
- Mixing code with conditions and loops
- Running code when a variable changes
- Dealing with accidents and the unexpected

In chapter 2 we 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 engender a more immediate understanding as we encounter conditions, loops, functions, classes, and the like.

As mentioned in chapters 1 and 2, JavaFX Script is what's referred to as an *expression language*, meaning most executable bits of code return either zero or 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 progress through this chapter, particularly if you're not used to languages working this way. I'll point out the ramifications as the chapter unfolds.

We'll 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 that 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. First, source code is presented in small chunks that (unless otherwise stated) compile and run independently, with the console output presented in bold. Second, to avoid too much repetition I'm continuing to incur the wrath of pedants by occasionally referring to the language variously as JavaFX Script (its proper name) or by the shorthand JavaFX or JFX.

We have quite an exciting journey ahead, with a few twists and turns, so let's get going. We'll start by looking at the highest level of structure we can apply to 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 variables and functions (see *access modifiers*, later), and to permit classes to have identical names but different purposes. Listing 3.1 shows an example.

Listing 3.1 Using the `package` statement to shorten class names

```
import java.util.Date;

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

Date lives in the package java.util

Here we see two different ways of creating a `Date` object. 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. Listing 3.2 shows how we can create our own packages.

Listing 3.2 Including a class inside a package

```
package jfxia.chapter3;

public class Date {
    override function toString() : String {
        "This is our date class";
    }
};
```

Should go in the file Date.fx

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.

In terms of how packages are physically stored, JavaFX Script uses the same combination of directories and class files as Java. 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 (OO), 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 JVM 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. The key difference is that JavaFX Script permits something called *mixin* inheritance, which offers more power than Java's interfaces but stops short of full-blown multiple-class inheritance. In the coming subsections we'll explore the ins and outs of JFX classes and how to define, create, inherit, control, and manipulate them.

> **Changes to object orientation**
>
> Starting with JavaFX 1.2, full-blown multiple inheritance was dropped from the JavaFX Script language in favor of *mixins*. The former proved to have too many *edge case* issues, while the latter is apparently much cleaner. We'll look at how mixins work later in this chapter.

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 names with source files. In JavaFX Script the source file 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 (see listing 3.3), the stuff that runs when your program starts. The code, in listing 3.3, lives in a file called "Examples2.fx".

Listing 3.3 Scripts and classes

```
package jfxia.chapter3;

def scriptVar:Integer = 99;    <-- Script variable
```

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

Script function

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

Bootstrap code

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

Listing 3.3 shows variables and functions in the script context. We saw plenty of examples of this type of code in chapter 2, 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 methods and 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 the class, but the script context can also contain loose code that 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 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 that will be explained later in this chapter.

Now that you've 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 its 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 they are accessed via a class instance (an object of the class type), so we'll stick to that terminology. To make the following prose flow more readily I'll sometimes refer to functions and variables using the combining term *members*, as in "script members" or "instance members."

Listing 3.4 defines a music track with three variables and two functions.

Listing 3.4 Class definition, with variables and functions

```
class Track {                                    <-- Track class
    var title:String;
    var artist:String;                                          Instance
    var time:Duration;                                          variables

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

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

var song:Track = Track {                                        Declaring a
    title: "Special"                                            new object
    artist: "Garbage"                                           from Track
    time: 220s
};
println(song);
```

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

Note that 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 that 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.

Functions in JFX classes are defined using the keyword `function`, followed by the function's name, a list of parameters, and their type in parentheses, followed by a colon and the return type (where `Void` means no return). There are two functions in listing 3.4: the first accepts another `Track` and checks whether 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. As with Java, all objects in JavaFX Script have a `toString()` function. Just

Properties: what's in a name?

Although the JavaFX Script documentation prefers the formal term *instance variables*, sometimes public class variables are referred to as *properties*. In programming, a property is a class element whose read/write syntax superficially resembles a variable, but with code that 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 instance 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#.

like Java, this function is called automatically whenever the compiler encounters an object in circumstances that require a `String`. So `println(song)` is silently translated by the compiler into `println(song.toString())`.

The keyword `this` can be used inside the class to refer to the current object, although its 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 via objects of the class. External access (outside the script) is controlled by access modifiers (discussed 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`. We didn't strictly need to specify the type after `song` (*type inference* would work), but since this is your first proper class example, I thought it wouldn't hurt to go that extra mile.

Before we move on, some bits and pieces need extra attention. Take a look at the source code in listing 3.5.

Listing 3.5 A closer look at functions

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

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

Second, and far more important, shouldn't `doesReturn()` have a `return` keyword in there somewhere? Recall that JavaFX Script is an expression language, and most of

its code constructs *give out* a result. Thus, the last expression of a function can be used for the return value, even if the `return` keyword itself is missing.

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 its distinctive declarative syntax.

By invoking what the JFX documentation snappily calls the *object literal* syntax, we can declaratively create new objects. Listing 3.6 shows how. Simply state the class name, open curly braces, list each instance variable you want to give an initial value to (using its name and value separated by a colon), and don't forget the closing curly braces. (The source file for listing 3.6 is SpaceShip.fx.)

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

```
package jfxia.chapter3;

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
};

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

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

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

Script variables

Our class, including init / postinit blocks

Declarative syntax

Again, declarative syntax

Only name set

Java-style syntax

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

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 semicolons.

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 as follows:

1. The virgin object is created.
2. The Java superclass default constructor is called.
3. The object literal's instance variable values are computed *but not set*.
4. The instance variables of the JavaFX superclass are set.
5. The instance variables of this class are now set, in lexical order.
6. The `init` block of each class in the object's class hierarchy is called, if present, starting from the top of the hierarchy. The object is now considered initialized.
7. The `postinit` block of each class in the object's hierarchy is called, if present, starting from the top of the hierarchy.

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 0.

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 could provide multiple means of configuring an object, like separate `lengthCm` and `lengthInches` variables, and detect which was used.

Moving on to the fourth example, you'll note that 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; however, we should resist that temptation. The `new` syntax should be used only when JavaFX Script's declarative syntax will not work (the example is *bad practice*!)

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.

JavaFX Script and semicolons

We touched on the issue of semicolons in the previous text. As in Java, semicolons are used to terminate expressions, but the JavaFX Script compiler seems particularly liberal about when a semicolon is necessary. When we meet ternary expressions in section 3.3.3, we'll see an example of where they cannot be used, but other than that there's apparently little enforcement of where they *must* be used. It seems when the compiler can infer the end of a construct using a closing brace or whitespace alone, it is happy to do so.

For the purposes of this book's source code, I'm adopting a general style of adding in semicolons at appropriate places, even if they can be omitted. You can drop them from your own code if you want, but I'm keeping them in the sample code to provide both clarity and familiarity.

3.2.4 Object declaration and sequences

There's nothing special about listing 3.7. It simply pulls together the object-creation syntax we saw previously with the sequence syntax we witnessed last chapter. It's worth an example on its own, as this mixing of objects and sequences crops up frequently in JavaFX programming.

Listing 3.7 creates two `Track` objects inside a sequence (note the `Track[]` type). To run this code you need the `Track` class we saw in listing 3.4.

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());
```

toString() could be omitted

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

Instead of a boring list of comma-separated numbers or strings, the sequence contains object literal 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 referenced as more than one type. (For readers unfamiliar with terms like *polymorphism*, there's a beginners' guide to object orientation in appendix C, section C5).

Java programmers may be surprised to learn that JavaFX Script deviates from the Java model for object orientation. As well as supporting Java's single inheritance, JavaFX Script supports *mixin inheritance*. We'll look at mixins a little later; first we need to get familiar with the basics of single inheritance from a JavaFX point of view. The following example (listings 3.8, 3.9, and 3.10) 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;
    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}";
    }
}
```

Instance variables

Instance functions, new for this class

Instance function, inherited

Listing 3.8 shows a simple `Animal` class. It's home to just a life gauge 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 definition tells the compiler that objects of this class cannot be created directly. Sometimes we need a class to be only 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 actually creating an `Animal` object itself. We'll see an example of this shortly.

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

Listing 3.9 Class inheritance, part 2

```
class Mammal extends Animal {                    <-- Subclass of Animal
    override function getName() : String {
        "Mammal"                                 Overrides the
    }                                            one in Animal

    function giveBirth() : Mammal {
        var m  = Mammal { life:100 };            Brand-new
        m.born();                                instance
        return m;                                method
    }
}
```

The class overrides the `getName()` method to provide its own answer, which explains the `override` keyword prefixing the function. The extra keyword is required at the start of any function that overrides another; it doesn't do anything other than help document the code and make it harder for bugs to creep into our programs. If you leave 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 JVM 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, and then calls `born()`. The `born()` function is inherited from `Animal`, along with the `toString()` function.

So far, so good; how about another `Animal` subclass? Take a look at listing 3.10.

Listing 3.10 Class inheritance, part 3

```
class Reptile extends Animal {
    override var life =  200;                    <-- Override inherited initial value

    override function getName() : String {
        "Reptile"                                Overrides the
    }                                            function in Animal

    function layEgg() : Egg {
        var e = Egg {
            baby: Reptile {}                     Create a Reptile
        };                                       inside an Egg
        e;
    }
}
```

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

The Egg class itself

Again we have a subclass of `Animal`, this time called `Reptile`, 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 the 0 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 longhand (the Java way), first creating a `Reptile`, then creating the `Egg`, then plugging one into the other, but the JavaFX Script syntax allows us far more elegance.

Now finally we need code to test our new objects. Listing 3.11 does just that.

Listing 3.11 Virtual functions demonstrated

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

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

First output line

Second output line

Third output line

Fourth output line

Final output line

```
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
```

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 its subclasses, such as `Mammal`, as `Animal` objects. That's the power of polymorphism.

Here's a quiz question: 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`. 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 from that of the other `Reptile` object? Well, `layEgg()` never calls `born()`, so `birthDate` is null. This is what causes the null values when we print out the day and month. And because `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 this 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 that happen to be reading this.

3.2.6 Mixin inheritance (mixin)

Mixin, a portmanteau of *mix* and *in*, is the name of a *lightweight* inheritance model that complements the class inheritance we've already seen. In JavaFX Script each class can be subclassed from one parent at a time; this neatly sidesteps potential conflicts and ambiguities between multiple parents but creates a bit of a straitjacket. Mixins allow a class to acquire the characteristics of another type (a mixin class) without full-blown multiple inheritance.

> **Changes to JavaFX Script inheritance**
>
> Early JavaFX Script compilers supported full-blown multiple inheritance; however, JavaFX 1.2 heralded a shift toward mixins and introduced the `mixin` keyword to the language. The intent was to remove some of the edge-case complications and performance costs caused by multiple inheritance, while keeping much of its benefit.

Each JavaFX Script class can inherit at most one Java or JavaFX class but any number of Java interfaces and/or JavaFX mixins. In turn, each JavaFX mixin can inherit from any number of Java interfaces and/or JavaFX mixins. A mixin provides a list of functions and variables any inheriting class must have, plus optional default content. A class that inherits a mixin is called a *mixee* (because it's the recipient of a mixin).

The process works like this: if the mixee extends another regular class, the variables and functions (the class *members*) in this superclass are inherited as per usual, before the *mixing* process begins. Then each mixin class is considered in turn, as they appear on the `extends` list (we'll look at the syntax in a moment). If the *mixee* omits a function or a variable, rather than flag a compiler error, the missing content is automatically copied in ("mixed in") to the mixee, almost as if it had been cut and pasted. Mixin variables and functions can also carry default content—initial values and function bodies—and that too will be mixed in if absent from the mixee.

This is a lot to take in at once, so let's work through the process by way of an example. Listing 3.12 shows code that might be used in a role-playing game.

Listing 3.12 Mixins

```
mixin class Motor {
    public var distance:Integer = 0;

    public function move(dir:Integer,dist:Integer) : Void {
        distance+=dist;
    }
}

mixin class Weapon {
    public var bullets:Integer
        on replace { println("Bullets: {bullets}"); }
    }

    init {
        reload();
    }

    public function fire(dir:Integer) : Void {
        if(bullets>0)  bullets--;
    }

    public abstract function reload() : Void;
}

class Robot extends Motor,Weapon {
    override var bullets = 1000
        on replace { println("Bang"); }

    override function reload() : Void {
        bullets=100;
    }
}

class Android extends Robot , Weapon,Motor {
}
```

Mixin for movement

Mixin for weapons

Inherits both mixins

Weapon/Motor are redundant

Here we have two mixin classes and two *regular* classes. The `Robot` and `Android` classes might be character types, while the `Motor` and `Weapon` mixins might represent traits characters can have. As you can see, the `mixin` keyword prefixes a class definition to flag it as a mixin. Mixins are then inherited by listing them after the `extends` keyword, separated by commas. To implement mixin content in a mixee, we use the same syntax as for overriding class members in regular inheritance, including the `override` keyword.

Using the code in listing 3.12 for reference, the process of mixing would be as follows:

1. The `Robot` class does not inherit from any superclass, so we start from a clean slate, so to speak.
2. Because `Motor` appears first on the `extends` list, it's first to be considered. Our `Robot` class implements neither a `distance` variable nor a `move()` function, so

the default implementations are inherited, including their value/body. (If you can't be bothered to provide your own, you get one for free!)

3. The `Weapon` mixin is considered next. It specifies one variable and two functions: default content is provided for `bullets` and `fire()`, but `reload()` is abstract, forcing any potential mixee to provide its own body implementation. The `Robot` class provides implementations of `bullet` and `reload()`, but `fire()` is absent, so `Weapon`'s default is inherited.
4. `Robot` is complete, with the content of `Robot`, `Motor`, and `Weapon` mixed to create one class. The result can be successfully cast to any of these three types.
5. The `Android` class extends `Robot`, and as such its inheritance of `Weapon` and `Motor` has no effect—all the necessary members are already present. Even changing the order on the `extends` line cannot alter anything. `Android` should probably be amended to remove `Motor` and `Weapon` from its `extends` list—although they don't do any harm, their presence is potentially confusing.

You'll note from the code that mixins can contain `init` (and `postinit`) blocks, and their variables also support `on replace` triggers (we haven't looked at triggers yet, so you might want to bookmark this page and come back to it). How do these work when the mixee is used?

In the case of `init` blocks, when any object is created its class's superclass is initialized first (which in turn initializes its own superclass, creating a chain reaction up the class hierarchy), and then the class itself is initialized. Initialization involves the `init` blocks of each mixin being run in the order in which they appeared on the `extends` list, and then the class's own `init` block being run. The `postinit` blocks are then run, using the same order (mixins first, class last).

The effect for `on replace` blocks is similar, with triggers farther up the inheritance tree running before those in the class itself. In our example both `Weapon.bullets` and `Robot.bullets` have a trigger block (the latter complements, rather than overrides, the former). When `bullets` is assigned, the `on replace` block for `Weapon.bullets` runs first, followed by the one in `Robot.bullets`.

We can quickly mop up some of the remaining mixin questions with a mini-FAQ:

- *Q*: Can I declare an object using a mixin class?
  A: Not directly—mixins are effectively abstract. But you can create the object using its regular class type and then cast to one of its mixin types.
- *Q*: What's the relationship between mixins and Java interfaces?
  A: Java's interfaces are effectively viewed as mixins with only abstract functions.
- *Q*: Do I have to fully implement all the members of every mixin I inherit?
  A: No, but if any of the functions still have no bodies after the inheritance process (they are still abstract), the resulting class must be declared abstract. (Mixin classes themselves are, effectively, abstract.)
- *Q*: Can I use the `super` and `this` keywords in a mixin function body?
  A: Yes. They work much like they do with regular classes.

- *Q*: Can I use `super` in a mixee, in reference to a mixin's default function body?
  A: Yes. If you provide your own function body, you can still reference the default one you overrode from the mixin using `super`.
- *Q*: What if mixin member names clash with existing mixee member names?
  A: If they are compatible with the existing members, then nothing happens—they're considered to already be inherited. But if they are incompatible (like a variable having a different type than its namesake), a compiler error results.
- *Q*: What if two mixins have identical members, with different defaults?
  A: Mixin inheritance follows the order of the extends list—the earliest mixin wins!
- Q: If two mixins have identical functions, which does `super` reference?
  A: When a mixee provides its own implementation of a function that appears in more than one of its mixins, using `super` *to reference the original isn't sufficient. To specify precisely which mixin we're referring to, we can use its class name instead, thus,* `Eggs.scramble()` *and* `SecretMessage.scramble()`.

That concludes our look at class inheritance; you may wish to revisit it (particularly the stuff on mixins) once you've read through the remainder of this chapter, but for now let's push on with our exploration of the JavaFX Script language.

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 in this chapter) 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 parameters. Listing 3.13 shows a simple example.

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 on 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) and then a colon and the return type. In listing 3.13 the type is `function(:String):Boolean`, a function that accepts a single `String` and returns a

`Boolean`. We can assign to this variable any function that 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 (see listing 3.14). The receiving function uses a function signature for its parameters, just like the variable in listing 3.13.

Listing 3.14 Passing functions as parameters to other functions

```
function manip(s:String ,
    f:function(:String):String) : Void {
    println("{s} = {f(a)}");
}

function m1(s:String) : String {
    s.toLowerCase();
}

function m2(s:String) : String {
    s.substring(0,4);
}

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

Function with parameter function

Functions we pass into manip()

```
JavaFX = javafx
JavaFX = Java
```

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 that accepts a `String` and returns a `String`. Fortunately we happen to have two such functions, `m1()` and `m2()`, on hand. Both accept and return a `String`, with basic manipulation 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 the `String` and prints the result. A simple example, perhaps, yet one that adequately demonstrates the effect.

Being able to reference functions like this is quite a powerful feature. Imagine a list class capable of being sorted or filtered by a plug-in-able function, for example. But this isn't the end of our discussion. In the 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 application 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 that event happens. JavaFX Script's function types, with their ability to point at code, fit the bill perfectly. But having to create script or class functions for each event handler is a pain, especially because in most cases they're used in only one place. If only there were a shortcut syntax for one-time function creation. Well, unsurprisingly, there is. And listing 3.15 shows us how.

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) {
        println("Found {f.getName()}");
    }
};
walker.go();
```

Anonymous function

The class `FileSystemWalker` 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 that accepts a `java.io.File` object and returns nothing.

The most important function is `walk()`, which recursively walks a directory tree looking for files that end with the desired `extension`, calling whichever function has been assigned to `action` for each match, passing the 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`, an object of type `FileSystemWalker`, is created.

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 act only on filenames with that ending. But look at the way `action` is assigned.

Rather than pointing to a function elsewhere, the `action` 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, making it 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 variables.)

There are four basic modes of visibility in JavaFX Script, outlined in table 3.1.

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.
`public`	Visible to anyone, anywhere.

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

Table 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 their use is with a task-centric mini-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. Is this 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).

Listing 3.16 Access modifiers on a class

```
package jfxia.chapter3.access;

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

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

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

Basic modes

Additive modes

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

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 such as in listing 3.17.

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
➥   from outside package
    //sPackage: "set";

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

    sPublic: "set";

    // ** sPublicReadDefault has script only (default)
➥   initialization access
    //sPublicReadDefault: "set";

    // ** sPublicReadPackage has package initialization
➥   access
    //sPublicReadPackage: "set";

    sPublicInitProtected: "set";
};

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

def str:String = a.sPublicReadDefault;

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

Always works

Works during declaration

Fails outside declaration

Read is okay

Listing 3.17 tests the `AccessTest` class we saw in listing 3.16. It lives in a different package than `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 shown in a comment on the preceding line.

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 that 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. In the 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 render 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 following sections 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. It goes without saying that 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.

First let's look at some basic conditions.

3.3.1 Basic conditions (if, else)

We'll kick things off with a basic example (listing 3.18).

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 this code, all depending on 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 that of countless other programming languages, but you'll recall mention of "interesting tricks"—let's look at listing 3.19.

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 that follows. It will acquire the value `"Positive"`, `"Negative"`, or `"Zero"`, depending on the outcome of the condition. How is this happening? Let's 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 thus can be used on the right-hand side of an assignment, or as part of a bind, or in 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 this 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 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 on the value of `isHex` either the first or the second formatting string will be applied to the number `12345`. Note the lack of curly braces and the absence of a closing semicolon in each block of the `if/else`. When used in a ternary sense, each part of the `if` construct should house just a single expression; semicolons and braces would be needed only for multiple expressions, which would not fit the ternary format. Any semicolon should therefore come at the end of the entire `if/else` construct.

Listing 3.21 shows 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
```

Here we see the true power of conditions being expressions. What amounts to a *switch* construct is actually directly providing the parameter for a function 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 requires 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, with listing 3.22.

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 )
1: flag=false, ambiguous=Hello (class java.lang.String)
```

Two different executions

Admittedly, when I first wrote this code, I expected a compiler error. None was forthcoming. This time we use not hard-coded values but a weak pseudorandom 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, 1 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 on `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 are strongly associated with 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's begin with listing 3.23; a basic example that introduces the `for` 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 we tie the loop to a sequence, defined by a range, and then pull each element out into the loop variable.

3.4.2 For loops as expressions (indexof)

Now for something that exploits the expression language facilities; check out listing 3.24.

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 on 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 0, the second is 1, 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 features 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 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 {
    public-init var name: String;
    public-init var players: String[];
}

var fiveAsideLeague:SoccerTeam[] = [
    SoccerTeam {
        name: "Java United"
```

A soccer team class

First soccer team object

```
        players: [ "Smith","Jones","Brown",
            "Johnson","Schwartz" ]
    } ,
    SoccerTeam {
        name: ".Net Rovers"
        players: [ "Davis","Taylor","Booth",
            "May","Ballmer" ]
    }
];

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

First soccer team object

Second soccer team object

Print players within teams

```
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: May
.Net Rovers: Ballmer
```

First we define our team class; then we create a sequence of two five-on-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 that walks over the teams, and we have the inner part that 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 by using a `continue`, and we force the loop to terminate with a `break` when it gets

beyond 10. In effect `total` is updated only 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 (see listing 3.27): 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 the example code runs over each element in a sequence from 0 to 50, but the added `where` clause filters out any loop value that isn't evenly divisible by 7. The result is the sequence `divisibleBy7`, whose contents we print. (Incidentally, in this particular example, since all we wanted to do was add the filtered elements into a new sequence, we could have used the predicate syntax we saw last chapter.)

That's it for sequence-based loops, at least for now. We'll again touch briefly on sequences 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. Cast an eye over listing 3.28.

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 are shown in listing 3.28, the first a simple loop, the second involving `break` and `continue` logic. To create a `while` loop we use the keyword `while`, followed by a terminating condition in parentheses and the body of the loop in curly braces. The first loop 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 latter causes the loop to be aborted. The result is that only the values 5 to 10 are totaled, also giving the answer 45.

As with `for` loops, `break` statements do not support a label pointing to which loop (of several nested) to break out of.

3.6 Acting 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 that can greatly aid us in creating sophisticated code that reacts to data change.

3.6.1 Single-value triggers (on replace)

Listing 3.29 demonstrates a simple trigger in action.

Listing 3.29 Trigger on variable change

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

Runs when current is changed

```
current=99 previous=0
current=7 previous=99
current=-8 previous=7
```

The `TestTrigger` class 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 sign`, and a variable to hold the replacement value. In the 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, not an assignment. I suppose `oldVal` and `newVal` could have been separated by a comma, but the language designers presumably thought the equals sign 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`.

For convenience, the trigger syntax can be abbreviated, as shown in listing 3.30.

Listing 3.30 Shorter trigger syntax

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

No old or new value

Old value only

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

Initialization

This shows two shorter syntax variants. The first doesn't bother with either the `newVal` or the `oldVal`, while the second bothers only with the `oldVal`.

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 previous and replacement values but also the range of the sequence that's being affected. Fortunately we can use a trigger itself to demonstrate how it works, as listing 3.31 proves.

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 tells the story of how the trigger was used. The `on replace` syntax has been complemented 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 preinsert 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 it 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 are no new values in a delete, obviously), and `firstIdx` and `lastIdx` describe the index range of the elements being removed.

Finally we repopulate the sequence with fresh data, causing an insert trigger, and then reverse the sequence to cause a mass replacement. Note how during the reverse the `firstIdx` and `lastIdx` values actually express 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 that's hard to understand and a nightmare to debug. And speaking of code that doesn't do what we think it should, in the next section we look at exceptions (talk about a slick segue!)

3.7 Trapping 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 that may be using our API. As always, we begin with an example. Take a look at listing 3.32.

Listing 3.32 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!
This always runs
```
key = 0

```
ERROR reading data java.io.IOException: Data corrupt
This always runs
```
key = 1

```
ERROR unknown fault
This always runs
```
key = 2

The code 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 `try` block is the code we want to trap exceptions on, and the `catch` blocks are executed if the `doSomething()` function actually throws an exception. The first block will be activated if the function throws an `IOException`. The second uses the `any` keyword to trap other exceptions that might be thrown. And last, the `finally` block will always be executed, regardless of whether or not an exception occurred.

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 catchall 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, such as closing a file properly before leaving a function. 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. Its 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 important, they permit 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 you feel your head is 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 you've just learned about.

In the next chapter we're not going to jump straight in with animations and swish UIs; 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, brew yourself a fresh cup of coffee, and I'll see you in the next chapter.