2

JavaFX Script ***DATA AND VARIABLES***

This chapter covers

- Introducing JavaFX Script variables
- Doing interesting things to strings
- Getting linear, with ranges and sequences
- Automating variable updates using binds

If chapter 1 has had the desired effect, you should be eager to get your hands dirty with code. But before we can start dazzling unsuspecting bystanders with our stunning JavaFX visuals, we'll need to become acquainted with Java FX Script, JavaFX's own programming language. In this chapter, and the one that 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, so in the next chapter we can explore 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. This logical progression wasn't always possible, and occasionally later detail bleeds 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 is. 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 term *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 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. There's not a lot to cover, because JavaFX Script uses the same C++-like syntax as many other popular languages (see listing 2.1).

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.

JavaFX Script also supports the familiar Javadoc comment format. These are specially formatted comments, written directly above class, variable, or function definitions, that can be turned into program documentation via a tool called javafxdoc. JavaFXDoc comments are multiline and begin with `/**`, the extra asterisk signaling the special JavaFXDoc format. All manner of documentation details can be specified inside one of

Give reasons for your answer

There's undoubtedly a certain resistance to source code commenting among programmers. It was ever thus! Comments acquired a bad name during the 1970s and 1980s, 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.

these comments; the available fields are listed on the OpenJFX project site (details of which can be found in appendix A).

Now that you know how to write code the compiler ignores, let's move on to write something that 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 on animation, it also features a type to represent time.

JavaFX Script's approach to variables is slightly different than that of Java. Java employs a dual strategy, respecting both the *high-level* objects of object-orientation and the *low-level* primitives of bytecode, JavaFX Script has only objects. But 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 on it. Strings will not, for example, magically turn themselves into numbers so we can perform arithmetic on them, even if these strings contain only valid number characters. In that regard they work the same way as the Java language.

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

2.2.2 Value type declaration

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 that stems from using classes.

Table 2.1 JavaFX Script value 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
`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`

One thing making value types stand out from their object brethren is that 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 source is implemented by familiar constructors and methods, but the string syntax keeps source code readable.

Another difference between value types and other objects is that 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.

Many of the types in table 2.1 are recognizable as variations on the basic primitive types you often find in other programming languages. The `Duration` type stands out as being unusual. It reflects JavaFX Script's focus on animation and demands more than just a basic explanation, so we'll put it aside until 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`. 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 example code defining the 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 semicolon 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 accept them—we'll deal with the details in a few pages.

As in many languages, the initializer is optional. We could have ended each declaration after its type, with just closing semicolons, resulting in each variable being initialized to a sensible default value. Listing 2.3 shows a handful of examples.

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 listing 2.4 shows:

Listing 2.4 Defining value types using type inference

```
var infBool = true;
var infFlt = 1.0;          <— Be careful declaring non-ints
var infInt = 1;
var infStr = "Some text";
println("Infered: B: {infBool}, "
    "F: {infFlt}, I: {infInt}, S: {infStr}");
println("{infFlt.getClass()}");
```

```
Infered: B: true, F: 1.0, I: 1, S: Some text
class java.lang.Float
```

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 `infFlt` 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 seen variables declared using the `var` keyword. But there's 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;                 Compiler error if
//cannotAssign = 55;        <—  uncommented
```

Using `def` instead of `var` results in variables that cannot be reassigned once created.

It's tempting to think of def variables only as constants; indeed that's how they're often used, but it's not always the case. A def variable cannot be reassigned, but the object it references can mutate (change its contents). Some types of objects are immutable (they provide no way to change their content once created, examples being String and Integer), so we might assume a def variable of an immutable type *must* be a constant. But again, this is not always the case. In a later section we'll investigate bound variables, revisiting def to see how a variable (even of an immutable type) 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?" First, def is useful if we want to drop hints to fellow programmers as to how a given variable should be used. Second, 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 it's 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 those of most programming languages. Unlike Java, 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, in listing 2.6.

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();                    <- Converting nl to an integer
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
```

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 later—for now be aware that 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 is a list of the common arithmetic operators; let's see some of them in action (see listing 2.7).

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();      <| Number literals
println("int1 = {int1}, "                      are objects too
    "int2 = {int2}, int3 = {int3}, num = {num}");
```

```
int1 = 20, int2 = 50, int3 = 1, num = 0.5
```

It's all very familiar. The only oddity in listing 2.7 is the value 2.5 having a method called on 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 `Float` type, allowing function calls on it.

2.2.5 Logic operators (and, or, not, <, >, =, >=, <=, !=)

In the next chapter we'll look at code constructs, such as conditions. But, as we're currently exploring the topic of data, we might as well take a quick gander (courtesy of listing 2.8) at the logic operations that form the backbone of condition code.

For most programmers, apart from a few differences in keywords (`and` and `or` instead of `&&` and `||`), there's no great surprise 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?

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();    <-- Creating a Java Date object
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
```

2.2.6 Translating and checking types (as, instanceof)

Because JavaFX Script is a statically typed language, casts may be required to convert one data type into another. If the conversion is guaranteed to be safe, JavaFX Script allows it without a 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 that needs converting. Listing 2.9 shows an example.

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 smaller one, as in the example, 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 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 if 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 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 doesn't matter which you use. Listing 2.11 shows even more string syntax variations.

Listing 2.11 Multiline 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 when on separate source code lines. 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 shown 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 is 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 `rating` in braces. Escaping the opening and closing braces with a backslash prevents any attempted expression evaluation, as has happened with `eval2`.

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 on 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 shows the JFX equivalent.

Listing 2.13 String formatting

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

def magic = -889275714;
println("{magic} in hex is {%08x magic}");            <- Eight-digit hexadecimal

def cal:Calendar = Calendar.getInstance();
cal.set(1991,2,4);
def joesBirthday:Date = cal.getTime();
println("Joe was born on a {%tA joesBirthday}");      <- Display date's weekday
```

```
-889275714 in hex is cafebabe
Joe was born on a Monday
```

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 representation of the value of `magic`. The second example creates a date for 4 March 1991 using Java's `Calendar` and `Date` classes (months are indexed from 0, while days are indexed from 1). 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, JavaFX Script uses a 32-bit signed `Integer` type, 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. Because 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 store the number 3405691582 in an integer, the compiler would inform us it's too big. However -889275714 results in exactly the same 32-bit pattern.

In case you didn't know, CAFEBABE is the 4-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 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 programmers a whole load of pain if everyone would 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 its 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 is a simple two-line localization property file for UK English. Its filename and location are important.

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

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

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 lowercase 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 uppercase characters, documenting a specific country. Finally, the extension .fxproperties must be added.

In the book's source code the script for testing listing 2.14 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 runtime. Listing 2.15 shows 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
```

The listing demonstrates 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 `"Rubbish"` 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 loaded, the `"Trash"` string is used as a default.

The second line does 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 localization 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 are a couple of examples:

```
-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 shows it in action.

Listing 2.16 Declaring `Duration` types

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

Milliseconds, seconds, minutes, and hours

Output is milliseconds

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

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 times expressed using the literal notation: 25 milliseconds, 30 seconds, 15 minutes, and 2 hours. 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 shows a few examples of what can be done, using Java's `System.printf()` method to add platform-specific line separators into the output string.

Listing 2.17 Arithmetic on duration types

```
import java.lang.System;
def dur1 = 15m * 4;
def dur2 = 0.5h * 2;
def flag1 = (dur1 == dur2);
def flag2 = (dur1 > 55m);
def flag3 = (dur2 < 123ms);
System.out.printf(
    "dur1 = {dur1.toMinutes()}m , "
    "dur2 = {dur2.toMinutes()}m%n"
    "(dur1 == dur2) is {flag1}%n"
    "(dur1 > 55m) is {flag2}%n"
    "(dur1 < 123ms) is {flag3}%n");
```

15 minutes times 4

Half an hour times 2

Converted to minutes

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

Both `dur1` and `dur2` are created as `Duration` objects, set to one hour. The first is created by multiplying 15 minutes by 4, and the second is created by multiplying half an hour by 2. 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 we still have a lot to explore before we get there, for example "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 for 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. Listing 2.18 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 square-brackets syntax.
- When one sequence is used inside another, it is expanded in place. Sequences are always linear; multidimensional sequences are not supported.
- Sequences are equal if they are the same size and each corresponding element is equal; in other words, the same values in the same order. 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.
- Sequence type notation is the plain object type followed by square brackets. 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. Check out listing 2.20.

Listing 2.20 Sequence creation using a range

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

Two dots create a range

```
seq3[0] = 1,seq3[11] = 12, seq3[89] = 90
```

The sequence 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! Take a look at listing 2.21.

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()}");
```

Compiler warning under JavaFX 1.2

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

Listing 2.21 shows ranges created using an extra `step` parameter. The first runs from 0 to 100 in steps of 5 (0, 5, 10, ...), and the second does the same in reverse (100, 95, 90, ...) The third goes from 0 to 100 backwards, resulting (quite rightly!) in an empty sequence. Finally, just to prove ranges aren't all about integers, we have a range from 0 to 1 in steps of a quarter.

Ranges can nest inside larger declarations, expanding in place to create a single sequence. We can exploit this fact for more readable source code, as listing 2.22 shows.

Listing 2.22 Expanding one sequence inside another

```
def blackjackValues = [ [1..10] , 10,10,10 ];      <-- Expanding a range inside
println("blackjackValues = "                            another declaration
    "{blackjackValues.toString()}");

blackjackValues = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10 ]
```

Listing 2.22 creates a sequence representing the card values in the game Blackjack: aces to tens use their face value, while picture cards (jack, queen, king) are valued at 10. (Yes, *I am* aware aces are also 11—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 that demand 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                          | Just the
slice2 = 0 to 9                           | start/end
slice3 = 95 to 100                        | values
slice4 = 95 to 99
```

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 11 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 leave 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 1. If you leave 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 next weapon in the sequence arsenal we'll look at (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 shows this in action.

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 because 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 listing 2.25.

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 example 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.

That's covers insert, but what about deleting from sequences? Here's how.

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 removes the entire sequence.

One final trick is the ability to reverse the order of a sequence, as shown here.

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 ]
```

The code in listing 2.27 merely flips the order of `seq3`, from 1 through 10 to 10 through 1.

All this inserting, deleting, and reversing may seem pretty useful, but perhaps some of you are worried about how much processing power it takes to chop and change large runs of data during the course of a program's execution. Because sequences aren't merely simple arrays, the answer is "surprisingly little."

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 in listing 2.27 no data was actually rearranged!

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

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 binding, perhaps JavaFX Script's most powerful syntax feature.

2.6 Autoupdating 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 on. Although binding has many applications, it's particularly useful in UI 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 that links model and UI usually verbose, but quite often it lives miles away from either. Binding connects the interface directly to the source data or (more accurately) to a nugget of code that interprets the data. The code that 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. Listing 2.28 is a straightforward example:

Listing 2.28 Binding into a string

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

This is a for loop

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

This simple example 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 that follows is a bound *expression*. Expressions are bits of code that 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 reevaluates its contents each time the variable it depends on 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`? Technically it doesn't ever change. Its value changes, true, but its *real* content (the expression) never actually gets reassigned. This is why, back when we covered `var` and `def`, I warned against thinking of `def` variables as simple constants, even if their type is immutable. Because a one-way bound variable should not be *directly* assigned to, using `def` is more correct than using `var`.

Bind works not only with strings but other data types too, as we'll see in listing 2.29.

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 recalculate the expression associated with their binding, meaning they will always be one year behind or ahead of 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! Check out listing 2.30.

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

How do you use the `bind` syntax with a sequence? As luck would have it, that's the next bit of example code (listing 2.31).

Listing 2.31 Binding against a sequence element

```
var range = [1..5];                 <-- 'range' is [1,2,3,4,5]
def reference = bind range[2];
println("range[2] = {reference}");
delete range[0];                       <-- 'range' is [2,3,4,5]
println("range[2] = {reference}");
delete range;                          <-- 'range' is empty
println("range[2] = {reference}");
```

```
range[2] = 3
range[2] = 4
range[2] = 0
```

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 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 has the answer.

Listing 2.32 Binding to a sequence itself

```
var multiplier:Integer = 2;
var seqSrc = [ 1..3 ];
def seqDst = bind for(seqVal in seqSrc) { seqVal*multiplier; }
println("seqSrc = {seqSrc.toString()},"
    " seqDst = {seqDst.toString()}");
insert 10 into seqSrc;                          <— Change source sequence
println("seqSrc = {seqSrc.toString()},"
    " seqDst = {seqDst.toString()}");
multiplier = 3;                                 <— Change multiplier
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 ]
```

The code 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. (In JavaFX Script, `for` loops create sequences; we'll look at the syntax in detail next chapter.) 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 that 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? That's the first question we'll answer, using listing 2.33.

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 that uses an `if/else` block to control the update of the string `modeStatus`, resulting in the contents being either "on" or "off." The `if/else` block is reevaluated each time `mode` changes.

Listing 2.34 shows 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 {
    if(id.equals("adam")) { return "Adam Booth"; }
    else if(id.equals("dan")) { return "Dan Jones"; }
    else { return "Unknown"; }
}

function getStatus(id:String) : String {
    if(id.equals("adam")) { return "Online"; }
    else if(id.equals("dan")) { return "Offline"; }
    else { return "Unknown"; }
}
```

Functions, accept and return a string

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

We haven't covered functions yet, but it shouldn't be hard to figure out what the example code 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 on the `userID` passed in. 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 wanted only a simple bind that 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 shows how this can be achieved.

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
```

The code uses 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 do this only 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 listing 2.35 doesn't do the idea justice. Imagine a data structure whose 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 `dictionary` and `thesaurus` variables? (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 we use `var` instead of `def`.

2.6.7 The mechanics behind bindings

The OpenJFX project's own language documentation goes into some detail on how binding works and possible side effects that may occur if a particular set of circumstances conspire. You need to know two things when working with binds.

First, 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 that returns a value). This means, for example, you can't update the value of an outside variable (that is, one defined outside the bind) from within the body of code associated with a bind. This 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, 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 body. You cannot update the variable once declared, so you must use the `def` keyword. (The JavaFX Script 1.0 compiler allows both `var` and `def` keywords to be used, but the documentation appears to suggest only `def` will be supported in the future.)

Moving on to our second topic: when it comes to bound expressions, JavaFX Script attempts to perform something called a *minimal recalculation*, which sounds rather cryptic but means only it's as lazy as possible. An example (listing 2.36) 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 called only once. The result from the first half of the expression (the function call) was reused, because JavaFX knows it's not dependent on the value that 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.

2.6.8 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 thus any dependencies it may have are not factored into the bind recalculations. Listing 2.37 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, since 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 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 on, and these are included as triggers to cause a recalculation of the bind.

When writing your own API libraries, you need to be aware of whether the return value of each function is entirely self-contained or dependent on some external data (perhaps a class instance variable?). If data other than the function parameters could influence the return value, you should consider marking the function as bound.

2.6.9 Bound object literals

Now that 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 and introduce something we won't cover fully until chapter 3. Object literals are JavaFX Script's way of declaratively creating complex objects. You should still be able to follow the basics of what follows, in listing 2.38, although you may want to bookmark this section and revisit it once you've read chapter 3.

Listing 2.38 Binding and object literals

```
class TextContainer {     <-- Create a class
    var label:String;
    var content:String;
    init {
        println("Object created: {label}");
    }
}

var someContent:String = "Pew, Pew, Barney, Magreu";     <-- Some text to bind to

def obj1 = bind TextContainer {     <-- Causes output line 1
    label: "obj1";
    content: someContent;
};
def obj2 = bind TextContainer {     <-- Causes output line 2
    label: "obj2";
    content: bind someContent;
};

someContent = "Cuthbert, Dibble and Grub";     <-- Causes output line 3

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

In the example 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 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 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 is a subtle difference between the two declarations that will alter the way binding works.

Creating the two objects causes the `init` code to run, resulting in 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 re-created, 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 that 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 on changes.

If you don't yet appreciate how useful JavaFX bindings are, you'll get plenty of examples in our first project in chapter 4. In the next section we'll examine some bits of syntax that, although not as powerful as binds, can be just as useful.

2.7 Working nicely with Java

JavaFX Script is designed to work well with Java. As you'll see in the next chapter, Java classes can be brought directly into JavaFX programs and used with the same ease as JavaFX Script classes themselves. But inevitably there will be a few areas where the two languages don't mesh quite so easily. In the following sections we'll look at a couple of useful language features that help keep the interoperability cogs well oiled.

2.7.1 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. Because JavaFX sits atop the Java platform and has access to Java libraries, it's possible names originating in Java may clash with keywords or syntax patterns in JavaFX Script. Those peculiar angle brackets resolve 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, and so on) 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 listing 2.39 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" were something in Java we needed to reference.

2.7.2 Handling Java native arrays (nativearray of)

Alongside its array classes (like `Vector` and `ArrayList`) Java also has in built native arrays. Java native arrays are a fixed size and feature a square-bracket syntax similar to JavaFX Script's sequences. But native arrays and sequences are not quite the same thing, as you may have guessed if you've read section 2.5.6. When working with Java classes we may encounter native arrays, and we need some easy way of bringing them into our JavaFX programs. Listing 2.40 shows how it can be done.

Listing 2.40 Native arrays

```
def commaSep:java.lang.String = "one,two,three,four";
def numbers:nativearray of java.lang.String =
    commaSep.split(",");
println("1st: {numbers[0]}");
for(i in numbers) { println("{i}"); }
```

```
1st: one
one
two
three
four
```

Here we create a Java `String` object containing a comma-separated list and then use a `String.split()` call to break the list up into its constituent parts, resulting in an array of `String` objects. To bring this array into JavaFX Script as an actual array (rather than a sequence), we declare `numbers` as being of type `nativearray of String`. The `numbers` object can then be read and written to use the familiar square-bracket syntax and even feature as part of a `for` loop.

While this eases the process of working with Java native arrays, it's important to remember that a variable declared using `nativearray` is not a sequence; for example, it may contain `null` objects, and the `insert/delete` syntax from section 2.5.5 will not work.

Java strings versus JavaFX strings

There's no significance in declaring `commaSep` as a Java `String`. Under-the-hood JavaFX Script strings use Java strings, so either would have worked. However, because `split()` is a Java method, returning a Java native array, I thought the code would be more readable if I spelled out explicitly the Java connection.

WARNING *Experimental feature* The `nativearray` functionality was added to JavaFX 1.2 as an experiment. It may be modified, or even removed, in later revisions of the JavaFX Script language.

2.8 Summary

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

And speaking of animation, there was one small piece of the JavaFX Script's data syntax we missed 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 over discussion of this one small part of the language for a later chapter.

In the next chapter we'll complete our tour of the JavaFX Script language by looking at the meat of the language, the stuff that actually gets our data moving.