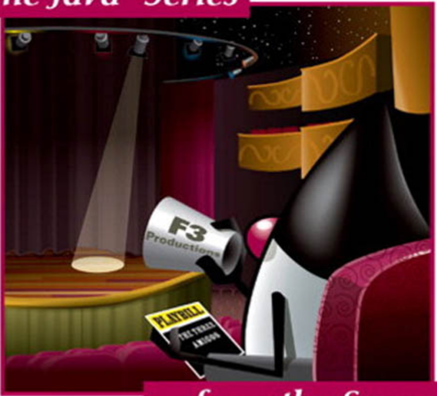
Jim Clarke · Jim Connors · Eric Bruno

JavaFX™

Developing Rich Internet Applications

Foreword by John Burkey

The Java™ Series



...from the Source

Library of Congress Cataloging-in-Publication Data:

Clarke, Jim.
JavaFX : developing rich internet applications / Jim Clarke, Jim Connors, Eric Bruno.
p. cm.
Includes index.
ISBN 978-0-13-701287-9 (pbk. : alk. paper)
1. Java (Computer program language) 2. JavaFX (Electronic resource) 3. Graphical user interfaces (Computer systems) 4. Application software Development. 5. Internet programming. I. Connors, Jim, 1962– II. Bruno, Eric J., 1969– III. Title.
QA76.73.J38C525 2009
006.7'6—dc22

2009014387

ISBN-13: 978-0-13-701287-9
ISBN-10: 0-13-701287-X
Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.
First printing, May 2009

Foreword

It is not often that you get the chance to witness (let alone participate in!) the birth of a truly disruptive technology. We are now at a juncture where information is pervasive—there is a convergence that will allow us to seamlessly move from one information source to another as we conduct our daily lives. Whether we are operating our smart phones, watching television, using our laptops, or interacting with screen-based devices that are yet to be invented, we are constantly connected to the world.

The key to making this vision a reality is the implementation of a common platform that works across all these screens. The Java platform set the bar for "write once, run anywhere"; JavaFX raises that bar by allowing us to write rich, immersive applications that run not only on every platform, but look good on every screen.

JavaFX is more than that, of course. It's about

- Employing visual effects to make the graphics stand out and appear real
- Adding animation to bring the screen to life
- Engaging the auditory and visual senses to more effectively convey information
- Combining all of these qualities to create compelling applications that are also fun to use

Of course, these capabilities are useless if applications cannot be crafted easily and quickly. Another goal of JavaFX is to make development simpler, easier, more productive—and more fun. The JavaFX script language was built from the ground up to support the scene-graph-based programming model, allowing the code to have a structure similar to the data structures it creates. Instead of looking for an esoteric “main” routine, the primary entry point is a “stage.” The stage has a “scene,” and “nodes” make up the elements in the scene. The analogy to the real world should be clear to all.

Second, the language supports, as a first class concept, the notion of *binding* between data elements. What used to take many lines of repetitive (and error-prone) listener code is now represented using a simple `bind` declaration. As a result, the display and your data model are automatically kept in sync, without having to write the many lines of code that would otherwise be required to connect them.

Lastly, the JavaFX platform provides a robust set of framework classes that allow you to quickly and simply exploit the most advanced features, such as animations, visual effects, and sophisticated visual transitions. All this adds up to a highly productive environment that allows you to quickly deploy the most advanced applications to both desktops and mobile devices in a fraction of the time.

Programmer productivity is only part of the story—rich applications also require participation from graphic designers and UI designers. JavaFX provides tools to integrate the graphic design process with the development process. For instance, the creative folks typically design the application’s look and feel, produce graphical assets, and then hand all of this over to the development team to create the program logic. The JavaFX Production Suite facilitates this handoff in an efficient way that allows developers and designers to collaborate easily.

When I joined the JavaFX project, I knew that I had embarked on a journey to create the best Rich Internet Application platform on the planet—a journey that has only just begun. I invite you to join this journey, with this book as your starting point. It begins with the basics and builds up to deploying a full-fledged application in JavaFX, covering all the features and capabilities that JavaFX provides along the way. Once you learn JavaFX, I’m sure you will be just as enthusiastic about this technology as I am. I welcome you aboard.

John Burkey
Chief JavaFX architect

Preface

Welcome to Rich Internet Application development with JavaFX.

This book is about creating more engaging user applications using special effects and animation. In this book, we will focus on using JavaFX for creating Rich Internet Applications.

Building upon the widely adopted and popular Java Platform, JavaFX provides a new level of abstraction that greatly simplifies graphical user interface development while at the same time bringing all the flexibility that Java technologies provide. This creates an elegant, yet powerful, platform for building full feature and compelling applications.

What Is JavaFX?

JavaFX is actually a family of products developed at Sun Microsystems. There are initiatives for mobile phones, consumer, television, and desktop devices. The cornerstone to these projects is JavaFX. JavaFX is a platform that includes a high performance declarative scripting language for delivering and building a new generation of Rich Internet Applications.

The primary focus of JavaFX is to make graphical user interface development easy while embracing more compelling features like visual effects, sound, and animation. JavaFX includes a ready-made framework to support graphic components and to easily include multimedia features like pictures, video, audio, and animation. Using the Java platform at its core, JavaFX works seamlessly with the Java platform and can easily leverage existing Java code. This also allows JavaFX to leverage the "write once, run anywhere" capability provided with the Java platform.

Why JavaFX?

Anyone who has ever written a graphical user interface application can appreciate the complexity of creating such an application. Though the resulting user interface can produce a powerful user experience, developing a cool application can be a daunting task. It takes a skilled developer who knows the graphical language and framework inside-out to pull off a well-written UI. JavaFX addresses this complexity.

Furthermore, graphic design and programming are two distinct skills. Graphic designers focus on the human interaction with the application, and are more interested in keeping the human's interest and making the system intuitive. On the other side, the program developers are typically concerned with implementing business logic and interacting with back-end servers. It is a rare breed that masters both of these skills. JavaFX's goal is to bridge these two crafts by allowing the graphic designer to dabble in an easily understood programming language, while at the same time allowing the developer the flexibility to implement the business rules behind the user interface.

JavaFX does this by

- Simplifying the programming language
- Providing ready-built user interface components and frameworks to support UI creations
- Making it easy to update existing UI applications
- Providing a cross-platform environment that delivers on "Write Once, Run Anywhere"

Rich Internet Applications

For many years, the programming paradigm has been centered on a client-server architecture employing a "thin" client. In this architecture, most of the processing was in the server with the client merely displaying the content. In a thin client system, data must be transmitted to the server for processing and a response sent back. This is very true of the HTML screens introduced with the original Internet browsers. However, by leveraging compute power on the client side, it is now possible to perform actions on the client, thereby reducing the round-trip latency to the server.

A Rich Internet Application is an application that allows a good portion of the application to execute on the user's local system. Primarily, the client application is designed to perform those functions that enhance the user's experience. Furthermore, communications with the server do not have to be initiated from a user action, like clicking on a button. Instead, a server itself can update the client with fresh content asynchronously as needed and without waiting for the end user to perform some action or by employing other tricks in the client like periodically polling the server.

So what is old is new again. In a sense this is true, but this really represents an evolution of the client server paradigm rather than a retrenchment back to the old days of the monolithic program that did everything. The key to a Rich Internet Application is striking the proper balance between behavior that should stay on the client with the behavior that rightfully belongs on the server. JavaFX is a framework that embraces the Rich Internet Application model.

Why This Book?

JavaFX is a new technology and we set out to help you get started quickly by exploring key features of JavaFX and how it should be used. We purposely did not want to do a language reference document as the language itself is fairly simple. Our main goal is to help you to quickly and productively create cool user interfaces.

This book's primary audience is comprised of developers (of all levels) and graphic designers who need to build Rich Internet Applications. There are different types of developers and designers that this book targets:

- Java developers who are currently building Rich Internet Applications with Java Swing
- Java developers who are interested in learning JavaFX for future projects
- Non-Java application developers who wish to use JavaFX for Rich Internet Application development
- Graphic designers, animators, or motion-graphic designers who wish to use JavaFX to add special effects, animation, and sound to their creations

How to Use This Book

This book has thirteen chapters. The first four chapters cover the basics of JavaFX, how to get started, what the graphic designer's role is, and the basic language. The next five chapters cover the advanced features you expect in a Rich Internet

Application. These include basic UI design, special effects, animation, multimedia, and browser display. Chapter 10 covers using JavaFX in a Web Services architecture. Chapter 11 describes JavaFX's interaction with the Java platform and assumes you are knowledgeable about Java. The last two chapters cover JavaFX code recipes and a complete Sudoku application.

Beyond the Written Page

With the expressive platform that JavaFX provides, it is hard to fully demonstrate all its capabilities on the written page. To fully appreciate all the features and capabilities that JavaFX brings, we suggest visiting the book's Web site http://jfxbook.com. There, you can see the full color versions of the figures used throughout the book. Also at the Web site, you can run the demos in full color and experience firsthand the richness of the animations and multimedia.

We have used a building block approach with basic concepts covered first and more complex features addressed later in the book, so we suggest you read each chapter in sequential order. If you are a graphic designer, you may be more interested in Chapter 2. You can safely start there, then jump back to Chapter 1 to dig deeper into JavaFX. If you are an "über"-coder, you can safely skip Chapter 2, but we still suggest you eventually read it just to know what the "dark" side is doing. Chapter 11 assumes you have a good understanding of the Java platform and APIs. If you do not plan to comingle your Java classes with JavaFX source in your application, you can safely skip this chapter. The last two chapters show some code examples based on the foundations laid down in the earlier chapters.

Here's the book in a nutshell:

- **Chapter 1: Getting Started.** This chapter gets you set up and shows the basics of creating and running a JavaFX program.
- **Chapter 2: JavaFX for the Graphic Designer.** This chapter explains how a graphic designer would use JavaFX to create JavaFX Graphical Assets.
- **Chapter 3: JavaFX Primer.** This chapter covers the basic JavaFX Script syntax.
- **Chapter 4: Synchronize Data Models—Binding and Triggers**. JavaFX Script introduces a data binding feature that greatly simplifies the model-view-controller design pattern. This chapter explains the concepts of data binding in the JavaFX Script language.
- **Chapter 5: Create User Interfaces**. The primary focus of JavaFX is to create rich user interfaces. This chapter explores the visual components

available to create user interfaces and demonstrates how the features of JavaFX work together to produce a rich user experience.

- **Chapter 6: Apply Special Effects**. A key to Rich Internet Applications is applying cool special effects to bring user interfaces alive and make them appealing to use. This chapter explores the special effects that JavaFX provides, including lighting, visual, and reflection effects.
- **Chapter 7: Add Motion with JavaFX Animation**. Animation makes the user interface vibrant and interesting. This chapter explains the concepts behind the JavaFX animation framework and provides examples of fade in/out, color animation, and motion. It also demonstrates an animation using Graphical Assets generated by the graphic designer.
- **Chapter 8: Include Multimedia**. This chapter explores how to include pictures, sound, and videos in your application.
- **Chapter 9: Add JavaFX to Web Pages with Applets.** (*Applets are back and these are not your father's applets*.) This chapter explores embedding JavaFX applications within Web pages and shows how to undock the applet from the Web page and demonstrate interaction with JavaScript.
- **Chapter 10: Create RESTful Applications**. JavaFX provides frameworks for working easily with JavaScript Object Notation (JSON) and Extensible Markup Language (XML). This chapter explores both options.
- **Chapter 11: JavaFX and Java Technology.** This chapter explores how JavaFX interacts with the Java platform.
- **Chapter 12: JavaFX Code Recipes**. Code recipes are general reusable solutions to common situations in programming. This chapter provides an overview of some code recipes applicable to programming JavaFX applications.
- **Chapter 13: Sudoku Application**. This chapter explores creating a Sudoku game application in JavaFX.

As we introduce topics, we have tried to inject our own experiences to help you avoid trial and error kinds of mistakes and "gotchas." Throughout the chapters, we have sprinkled Developer Notes, Warnings, and Tips to point out things that might not be obvious. We have also tried to include as many examples and figures as possible to illustrate JavaFX features and concepts.

This book is intended to cover the general deployment of JavaFX, whether it be on the desktop, mobile, or eventually the TV profiles. However, there is a bias toward the desktop version and specific features for JavaFX mobile are not covered. Still, the basic concepts and features covered in this book will also apply to these other profiles and to future releases of JavaFX.

Staying Up-to-Date

This book is written to the JavaFX 1.1 Software Development Kit (SDK). As this book goes to press, JavaFX 1.2 is being finalized. We have tried to include as many JavaFX 1.2 features as possible; however, not all features were fully defined in time. Please check out the book's Web site, http://jfxbook.com, for updates for the JavaFX 1.2 release.

This book is jam packed with demo and example code. To illustrate some features in print, we have abbreviated some of the examples. The complete code used in this book is available on the book's Web site at http://jfxbook.com. You can also check this site for updates, errata, and extra content. There is also a forum for sharing information about the book and JavaFX.

3

JavaFX Primer

"I'm still at the beginning of my career. It's all a little new, and I'm still learning as I go."

—Orlando Bloom

JavaFX Script Basics

JavaFX is partially a declarative language. Using a declarative language, a developer describes what needs to be done, then lets the system get it done. Olof Torgersson, Program Director for the Chalmers University of Technology Master's program in Interaction Design and Associate Professor at Göteborg University, has been researching declarative programming for over 10 years. From his analysis of declarative programming approaches, we find this definition:

> *"From a programmer's point of view, the basic property is that programming is lifted to a higher level of abstraction. At this higher level of abstraction the programmer can concentrate on stating what is to be computed, not necessarily how it is to be computed"*[1]

JavaFX Script blends declarative programming concepts with object orientation. This provides a highly productive, yet flexible and robust, foundation for applications. However, with this flexibility comes responsibility from the developer. JavaFX Script is a forgiving language and being declarative, it assumes inherent rules that may obscure a programming fault. The most obvious of these is that null objects are handled by the runtime engine and seldom cause a Java Null Pointer exception. As a result, the program will continue when a null is encountered

1. Torgersson, Olof. "A Note on Declarative Programming Paradigms and the Future of Definitional Programming," Chalmers University of Technology and Göteborg University, Göteborg, Sweden. http://www.cs.chalmers.se/~oloft/Papers/wm96/wm96.html.

within an expression, and will produce a valid result. However, the result may not have been what you expected. Therefore, the developer needs to be extra vigilant when writing code and more thorough when testing it. At first, this may seem alarming; however, this is offset by the ease of use and greater productivity of JavaFX and by the fact that JavaFX tries to mitigate the user from experiencing a crash.

One of the benefits of JavaFX being a declarative language is that much of the "plumbing" to make objects interact is already provided within the language. This allows the developer to be able to concentrate more on what needs to display, and less on how to do it. The next sections provide an overview of the JavaFX Script language including syntax, operators, and other features.

JavaFX Script Language

As we already mentioned, JavaFX Script is a declarative scripting language with object-oriented support. If you are already acquainted with other languages such as Java, JavaScript, Groovy, Adobe ActionScript, or JRuby, JavaFX Script will look familiar, but there are significant differences. While supporting traditional pure scripting, it also supports the encapsulation and reuse capabilities afforded by object orientation. This allows the developer to use JavaFX to produce and maintain small- to large-scale applications. Another key feature is that JavaFX Script seamlessly integrates with Java.

Conceptually, JavaFX Script is broken down into two main levels, script and class. At the script level, variables and functions may be defined. These may be shared with other classes defined within the script, or if they have wider access rights, they may be shared with other scripts and classes. In addition, expressions called *loose* expressions may be created. These are all expressions declared outside of a class definition. When the script is evaluated, all loose expressions are evaluated.

A very simple script to display Hello World to the console is

```
println("Hello World");
```

Another example, showing how to do a factorial of 3, is shown in Listing 3.1.

Listing 3.1 Factorial of 3

```
def START = 3;
var result = START;
var a = result - 1;
while(a > 0) {
    result *= a;
    a--;
}
println("result = {result}");
```

Developer Note: If your script has exported members—that is, any external accessible members such as `public`, `protected`, and `package`, functions or variables—then all *loose expressions* must be contained in a `run` function. For example, if we change the `result` variable in the previous example to add `public` visibility, we need to create the `run` function.

```
public var result:Number;

function run(args : String[]) : java.lang.Object {

    var num = if(sizeof args > 0) {
            java.lang.Integer.valueOf(args[0]);
        } else {
            10;
        };

    result = num;
    var a = result - 1;
    while(a > 0) {
        result *= a;
        a--;
    }
    println("{num}! = {result}");
}
```

The `run` method contains an optional `String[]` parameter, which is a sequence of the command-line arguments passed to the script when it runs.

If you do not have exported members, you can still include a `run` method. However, the `run` method, itself, is considered exported, even if you do not include an access modifier with it. So, once you add a `run` method, all loose exported expressions must now be contained within it.

Apart from the script level, a class defines instance variables and functions and must first be instantiated into an object before being used. Class functions or variables may access script level functions or variables within the same script file, or from other script files if the appropriate access rights are assigned. On the other hand, script level functions can only access class variables and functions if the class is created into an object and then only if the class provides the appropriate access rights. Access rights are defined in more detail later in this chapter.

Class Declaration

To declare a class in JavaFX, use the `class` keyword.

```
public class Title {
}
```

Developer Note: By convention, the first letter of class names is capitalized.

The `public` keyword is called an access modifier and means that this class can be used by any other class or script, even if that class is declared in another script file. If the class does not have a modifier, it is only accessible within the script file where it is declared. For example, the class `Point` in Listing 3.2 does not have a visibility modifier, so it is *only has script visibility* and can only be used within the `ArtWork` script.

Listing 3.2 Artwork.fx

```
class Point {// private class only
             //visible to the ArtWork class
    var x:Number;
    var y:Number;
}

public class ArtWork {
    var location: Point;
}
```

Developer Note: For each JavaFX script file, there is a class generated using that script filename, even if one is not explicitly defined. For example, in the previous example for `ArtWork.fx`, there is a class `ArtWork`. This is true even if we had not included the `public class ArtWork` declaration.

Also, all other classes defined within the script file have their name prepended with the script file's name. For example, in the previous example, class `Point` is fully qualified as `ArtWork.Point`. Of course, if `ArtWork` belongs to a package, the package name would also be used to qualify the name. For example, `com.acme.ArtWork.Point`.

To extend a class, use the `extends` keyword followed by the more generalized class name. JavaFX classes can extend at most one Java or JavaFX class. If you extend a Java class, that class must have a default (no-args) constructor.

```
public class Porsche911 extends Porsche {
}
```

JavaFX may extend multiple JavaFX `mixin` classes or Java interfaces. `Mixin` classes are discussed in the next section.

An application may contain many classes, so it is helpful to organize them in a coherent way called *packages*. To declare that your class or script should belong to a package, include a package declaration at the beginning of the script file. The following example means that the `Title` class belongs to the `com.mycompany.components` package. The full name of the `Title` class is now `com.mycompany.components.Title`. Whenever the `Title` class is referenced, it must be resolved to this full name.

```
package com.mycompany.components;
public class Title {
}
```

To make this resolution easier, you can include an import statement at the top of your source file. For example:

```
import com.mycompany.components.Title;

var productTitle = Title{};
```

Now, wherever `Title` is referenced within that script file, it will resolve to `com.mycompany.components.Title`. You can also use a wildcard import declaration:

```
import com.mycompany.components.*;
```

With the wildcard form of import, whenever you refer to any class in the `com.mycompany.components` package, it will resolve to its full name. The following code example shows how the class names are resolved, showing the fully qualified class name in comments.

```
package com.mycompany.myapplication;
import com.mycompany.components.Title;

// com.mycompany.myapplication.MyClass
public class MyClass {
    // com.mycompany.components.Title
    public var title: Title;
}
```

A class can have package visibility by using the `package` keyword instead of `public`. This means the class can only be accessed from classes within the same package.

```
package class MyPackageClass {
}
```

A class may also be declared `abstract`, meaning that this class cannot be instantiated directly, but can only be instantiated using one of its subclasses. Abstract classes are not intended to stand on their own, but encapsulate a portion of shared state and functions that several classes may use. Only a subclass of an abstract class can be instantiated, and typically the subclass has to fill in those unique states or behavior not addressed in the abstract class.

```
public abstract class MyAbstractClass {
}
```

If a class declares an `abstract` function, it must be declared `abstract`.

```
public abstract class AnotherAbstractClass {
    public abstract function
                  setXY(x:Number, y:Number) : Void;
}
```

Mixin Classes

JavaFX supports a form of inheritance called mixin inheritance. To support this, JavaFX includes a special type of class called a `mixin`. A `mixin` class is a class that provides certain functionality to be inherited by subclasses. They cannot be instantiated on their own. A `mixin` class is different from a Java interface in that the mixin may provide default implementations for its functions and also may declare and initialize its own variables.

To declare a `mixin` class in JavaFX, you need to include the `mixin` keyword in the class declaration. The following code shows this.

```
public mixin class Positioner {
```

A `mixin` class may contain any number of function declarations. If the function declaration has a function body, then this is the default implementation for the function. For example, the following listing shows a `mixin` class declaration for a class that positions one node within another.

```
public mixin class Positioner {
    protected bound function centerX(
                       node: Node, within: Node) : Number {
```

```
            (within.layoutBounds.width -
                                node.layoutBounds.width)/2.0 -
                                       node.layoutBounds.minX;
        }
        protected bound function centerY(node: Node,
                                      within: Node) : Number {
            (within.layoutBounds.height -
                             node.layoutBounds.height)/2.0 -
                                      node.layoutBounds.minY;
        }
    }
```

Subclasses that want to implement their own version of the `mixin` function must use the `override` keyword when declaring the function. For instance, the following code shows a subclass that implements its own version of the `centerX()` function from the `Positioner mixin` class.

```
    public class My Positioner extends Positioner {
        public override bound function centerX(node: Node,
                 within: Node) : Number {
            (within.boundsInParent.width -
                           node.boundsInParent.width )/2.0;
        }
    }
```

If the `mixin` function does not have a default implementation, it must be declared abstract and the subclass must override this function to provide an implementation. For instance, the following code shows an `abstract` function added to the `Positioner mixin` class.

```
    public abstract function bottomY(node: Node,
                 within: Node, padding: Number) : Number;
```

The subclass must implement this function using the `override` keyword, as shown in the following listing.

```
    public class My Positioner extends Positioner {
        public override function bottomY(node: Node,
                     within: Node, padding: Number) : Number {
            within.layoutBounds.height - padding -
                                    node.layoutBounds.height;
        }
    }
```

If two mixins have the same function signature or variable name, the system resolves to the function or variable based on which mixin is declared first in the extends clause. To specify a specific function or variable, use the mixin class name with the function or variable name. This is shown in the following code.

```
public class My Positioner extends Positioner,
                     AnotherPositioner {
    var offset = 10.0;
    public override bound function
             centerX(node: Node, within: Node) : Number {
        Positioner.centerX(node, within) + offset;
    }
}
```

Mixins may also define variables, with or without default values and triggers. The subclass either inherits these variables or must override the variable declaration. The following listing demonstrates this.

```
public mixin class Positioner {
    public var offset: Number = 10.0;
}

public class My Positioner extends Positioner {
    public override var offset = 5.0 on replace {
        println("{offset}");
    }
}
```

If a class extends a JavaFX class and one or more mixins, the JavaFX class takes precedence over the mixin classes for variable initialization. If the variable is declared in a superclass, the default value specified in the superclass is used; if no default value is specified in the superclass, the "default value" for the type of that variable is used. For the mixin classes, precedence is based on the order they are defined in the extends clause. If a variable declared in a mixin has a default value, and the variable is overridden without a default value in the main class, the initial value specified in the mixin is used.

Mixins may also have init and postinit blocks. Mixin init and postinit blocks are run after the super class's init and postinit blocks and before the subclass's init and postinit blocks. Init and postinit blocks from the mixin classes are run in the order they are declared in the extends clause for the subclass.

Object Literals

In JavaFX, objects are instantiated using *object literals*. This is a declarative syntax using the name of the class that you want to create, followed by a list of initializers and definitions for this specific instance. In Listing 3.3, an object of class `Title` is created with the text “JavaFX is cool” at the screen position 10, 50. When the mouse is clicked, the provided function will be called.

Listing 3.3 Object Literal

```
var title = Title {
      text: "JavaFX is cool"
      x: 10
      y: 50
      onMouseClicked: function(e:MouseEvent):Void {
         // do something
      }
};
```

When declaring an object literal, the instance variables may be separated by commas or whitespace, as well as the semi-colon.

You can also override abstract functions within the object literal declaration. The following object literal, shown in Listing 3.4, creates an object for the `java.awt.event.ActionListener` interface and overrides the abstract java method `void actionPerformed(ActionEvent e)` method.

Listing 3.4 Object Literal – Override Abstract Function

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

var listener = ActionListener {
    override function
             actionPerformed(e: ActionEvent) : Void {
        println("Action Performed!");
    }
}
```

Variables

JavaFX supports two kinds of variables: *instance* and *script*. Script variables hold state for the entire script, whereas instance variables hold state for specific instantiations of a class declared within the script file.

There are basically two flavors of variables: *unassignable* and *changeable*. Unassignable variables are declared using the def keyword and must be assigned a default value that never changes.

```
public def PI = 3.14;
```

These variables cannot be assigned to, overridden, or initialized in object literals. In a sense, these can be viewed as constants; however, they are not "pure" constants and can participate in binding. (For more information on binding, see Chapter 4, Synchronize Data Models—Binding and Triggers.)

Consider the following example of defining an *unassignable* variable that contains an object. The object instance cannot change, but that does not mean the state of that instance will not.

```
def centerPoint = Point{x: 100, y:100};
centerPoint.x = 500;
```

The actual Point object assigned to centerPoint remains unchanged, but the state of that object instance, the actual x and y values, may change. When used in binding though, centerPoint is constant; if the state of centerPoint changes, the bound context will be notified of the change.

Changeable instance variables are declared using the var keyword with an optional default value. If the default value is omitted, a reasonable default is used; basically, Numbers default to zero, Boolean defaults to false, Strings default to the empty string, Sequences default to the Empty Sequence, and everything else defaults to null.

Script variables are declared outside of any class declaration, whereas instance variables are declared within a class declaration. If a script variable is declared with one of the access modifiers—public, protected, or package—it may be used from outside of the script file, by referring to its fully qualified name. This fully qualified name is the combination of package name, script name, and the variable name. The following is the fully qualified name to a public script variable from the javafx.scene.Cursor class for the crosshair cursor.

```
javafx.scene.Cursor.CROSSHAIR;
```

Instance variables are declared within a class declaration and come into being when the object is created. Listing 3.5 illustrates several examples of script and instance variables.

Listing 3.5 Script and Instance Variables

```
import javafx.scene.Cursor;
import javafx.scene.paint.Color;

// import of script variable from javafx.scene.Cursor
import javafx.scene.Cursor.CROSSHAIR;

// Unchangeable script variable
def defaultText = "Replace ME"; // Script accessible only

// Changeable script variable
public var instanceCount: Integer; // Public accessible

public class Title  {
    // Unchangeable instance variables
    def defStroke = Color.NAVY; // class only access,
            //resolves to javafx.scene.paint.Color.NAVY

    // Changeable instance variables

    // defaults to the empty String ""
    public var text:String;
    public var width:  Number; // defaults to zero (0.0)
    public var height = 100; // Infers Integer type
    public var stroke: Color = defaultStroke;
    public var strokeWidth = 1.0; // Infers Number type
    public var cursor = CROSSHAIR;
            //resolves to javafx.scene.Cursor.CROSSHAIR

...
}
```

You may have noticed that some of the declarations contain a type and some don't. When a type is not declared, the type is inferred from the first assigned value. `String`, `Number`, `Integer`, and `Boolean` are built-in types, everything else is either a JavaFX or a Java class. (There is a special syntax for easily declaring `Duration` and `KeyFrame` class instances that will be discussed in Chapter 7, Add Motion with JavaFX Animation.)

Table 3.1 lists the access modifiers for variables and their meaning and restrictions. You will notice reference to initialization, which refers to object literal declarations. Also, you will notice variables being bound. This is a key feature of JavaFX and is discussed in depth in Chapter 4.

Table 3.1 Access Modifiers

Access Modifier	Meaning
`var`	The **default** access permission is script access, so without access modifiers, a variable can be initialized, overridden, read, assigned, or bound from within the script only.
`def`	The **default** access permission is script access; a definition can be read from or bound to within the script only.
`public` `var`	Read and writable by anyone. Also, it can be initialized, overridden, read, assigned, or bound from anywhere.
`public` `def`	This definition can be read anywhere. A definition cannot be assigned, initialized (in an object literal), or overridden no matter what the access permissions. It may be bound from anywhere.
`public-read` `var`	Readable by anyone, but only writable within the script.
`public-init` `var`	Can be initialized in object literals, but can only be updated by the owning script. Only allowed for instance variables.
`package` `var`	A variable accessible from the package. This variable can be initialized, overridden, read, assigned, or bound only from a class within the same package.
`package` `def`	Define a variable that is readable or bound only from classes within the same package.
`protected` `var`	A variable accessible from the package or subclasses. This variable can be initialized, overridden, read, assigned, or bound from only a subclass or a class within the same package.
`protected` `def`	Define a variable that is readable or bound only from classes within the same package or subclasses.
`public-read protected` `var`	Readable and bound by anyone, but this variable can only be initialized, overridden, or assigned from only a subclass or a class within the same package.
`public-init protected` `var`	Can be initialized in object literals, read and bound by anyone, but can only be overridden or assigned, from only a subclass or a class within the same package. Only allowed for instance variables.

You can also declare change triggers on a variable. Change triggers are blocks of JavaFX script that are called whenever the value of a variable changes. To declare a change trigger, use the `on replace` syntax:

```
public var x:Number = 100 on replace {
    println("New value is {x}");
};
public var width: Number on replace (old) {
    println("Old value is {old}, New value is {x}");
}
```

Change triggers are discussed in more depth in Chapter 4.

Sequences

Sequences are ordered lists of objects. Because ordered lists are used so often in programming, JavaFX supports sequence as a first class feature. There is built-in support in the language for declaring sequences, inserting, deleting, and modifying items in the sequence. There is also powerful support for retrieving items from the sequence.

Declaring Sequences

To declare a sequence, use square brackets with each item separated by a comma. For example:

```
public def monthNames = ["January", "February", "March",
                         "April", "May", "June",
                         "July", August", "September",
                         "October", "November", "December"];
```

This sequence is a sequence of `Strings`, because the elements within the brackets are `Strings`. This could have also been declared as

```
public def monthNames: String[] = [ "January", .....];
```

To assign an empty sequence, just use square brackets, `[]`. This is also the default value for a sequence. For example, the following two statements both equal the empty sequence.

```
public var nodes:Node[] = [];

public var nodes:Node[];
```

When the sequence changes, you can assign a trigger function to process the change. This is discussed in depth in the next chapter.

A shorthand for declaring a sequence of `Integers` and `Numbers` uses a range, a start integer or number with an end. So, `[1..9]` is the sequence of the integers from 1 thru 9, inclusive; the exclusive form is `[1..<9]`—that is, 1 through 8. You can also use a step function, so if, for example, you want even positive integers, use `[2..100 step 2]`. For numbers, you can use decimal fractions, `[0.1..1.0 step 0.1]`. Without the step, a step of 1 or 1.0 is implicit.

Ranges may also go in decreasing order. To do this, the first number must be higher than the second. However, without a negative step function, you always end up with an empty sequence. This is because the default step is always positive 1.

```
var negativeNumbers = [0..-10]; // Empty sequence
var negativeNumbers = [0..-10 step -1]; // 0,-1,-2,...-10
var negativeNumbers = [0..<-10 step -1]; // 0,-1,-2,...,-9
```

To build sequences that include the elements from other sequences, just include the source sequences within the square brackets.

```
var negativePlusEven = [ negativeNumbers,  evenNumbers ];
```

Also, you can use another sequence to create a sequence by using the Boolean operator. Another sequence is used as the source, and a Boolean operator is applied to each element in the source sequence, and the elements from the source that evaluate to true are returned in the new sequence. In the following example, `n` represents each item in the sequence of positive integers and `n mod 2 == 0` is the evaluation.

```
var evenIntegers = positiveIntegers[n | n mod 2 == 0];
```

One can also allocate a sequence from a `for` loop. Each object “returned” from the iteration of the `for` loop is added to the sequence:

```
// creates sequence of Texts
var lineNumbers:Text[]  = for(n in [1..100]) {
      Text { content: "{n}" };
};

// creates Integer sequence, using indexof operator
var indexNumbers = for(n in nodes) {
      indexof n;
};
```

To get the current size of a sequence use the `sizeof` operator.

```
var numEvenNumbers = sizeof evenNumbers;
```

Accessing Sequence Elements

To access an individual element, use the numeric index of the element within square brackets:

```
var firstMonth =  monthNames[0];
```

You can also take slices of sequence by providing a range. Both of the next two sequences are equal.

```
var firstQuarter = monthNames[0..2];

var firstQuarter = monthNames[0..<3];
```

The following two sequences are also equal. The second example uses a syntax for range to indicate start at an index and return all elements after that index.

```
var fourthQuarter = monthNames[9..11 ];

var fourthQuarter = monthNames[9.. ];
```

To iterate over a sequence, use the `for` loop:

```
for( month in monthNames) {
    println("{month}");
}
```

Modifying Sequences

To replace an element in a sequence, just assign a new value to that indexed location in the index.

```
var students = [ "joe", "sally", "jim"];
students[0] = "vijay";
```

Developer Note: As we said at the beginning of this chapter, JavaFX is a forgiving language, so if you assign to an element index location outside of the existing size of the sequence, the assignment is silently ignored.

Let's use the students sequence from the previous example:

```
students[3] = "john";
```

The assignment to position 3 would be ignored because the size of students is currently 3, and the highest valid index is 2. Similarly, assignment to the index -1 is silently ignored for the same reason; -1 is outside of the sequence range.

Furthermore, if you access an element location outside of the existing range for the sequence, a default value is returned. For `Numbers`, this is zero; for `Strings`, the empty string; for `Objects`, this is null.

To insert an element into the sequence, use the `insert` statement:

```
// add "vijay" to the end of students
insert "vijay" into students;

// insert "mike" at the front of students
insert "mike" before students[0];

// insert "george" after the second student
insert "george" after students[1];
```

To delete an element, use the `delete` statement:

```
delete students[0]; // remove the first student
delete students[0..1]; // remove the first 2 students
delete students[0..<2]; // remove the first 2 students
delete students[1..]; // remove all but the first student
delete "vijay" from students;
delete students; // remove all students
```

Native Array

Native array is a feature that allows you to create Java arrays. This feature is mainly used to handle the transfer of arrays back and forth from JavaFX and Java. An example of creating a Java `int[]` array is shown in the following code.

```
var ints: nativearray of Integer =
                [1,2,3] as nativearray of Integer;
```

Native arrays are not the same as sequences, though they appear similar. You cannot use the sequence operators, such as `insert` and `delete`, or slices. However, you can do assignments to the elements of the array as shown in the following code:

```
ints[2] = 4;
```

However, if you assign outside of the current bounds of the array, you will get an `ArrayIndexOutOfBounds` Exception.

You can also use the `for` operator to iterate over the elements in the native array. The following code shows an example of this.

```
for(i in ints) {
    println(i);
}
for(i in ints where i mod 2 == 0) {
    println(i);
}
```

Functions

Functions define behavior. They encapsulate statements that operate on inputs, function arguments, and may produce a result, a returned expression. Like variables, functions are either script functions or instance functions. Script functions operate at the script level and have access to variables and other functions defined at the script level. Instance functions define the behavior of an object and have access to the other instance variables and functions contained within the function's declaring class. Furthermore, an instance function may access any script-level variables and functions contained within its own script file.

To declare a function, use an optional access modifier, `public`, `protected`, or `package`, followed by the keyword `function` and the function name. If no access modifier is provided, the function is private to the script file. Any function arguments are contained within parentheses. You may then specify a function return type. If the return type is omitted, the function return type is inferred from the last expression in the function expression block. The special return type of `Void` may be used to indicate that the function returns nothing.

In the following example, both function declarations are equal. The first function infers a return type of `Glow`, because the last expression in the function block is an object literal for a `Glow` object. The second function explicitly declares a return type of `Glow`, and uses the `return` keyword.

```
public function glow(level: Number) {
      // return type Glow inferred
      Glow { level: level };
}

public function glow(): Glow { // explicit return type
      return glow(3.0); // explicit return keyword
}
```

The `return` keyword is optional when used as the last expression in a function block. However, if you want to return immediately out of an `if/else` or loop, you must use an explicit return.

In JavaFX, functions are objects in and of themselves and may be assigned to variables. For example, to declare a function variable, assign a function to that variable, and then invoke the function through the variable.

```
var glowFunction : function(level:Number):Glow;
glowFunction = glow;
glowFunction(1.0);
```

Functions definitions can also be anonymous. For example, for a function variable:

```
var glowFunction:function(level:Number): Glow =
    function(level:Number)  {
        Glow { level: level };
    };
```

Or, within an object literal declaration:

```
TextBox {
    columns: 20
    action: function() {
        println("TextBox action");
    }
}
```

Use `override` to override a function from a superclass.

```
class MyClass {
      public function print() { println("MyClass"); }
}
class MySubClass extends MyClass {
      override function print() { println("MySubClass"); }
}
```

Strings

String Literals

String literals can be specified using either double (") or single (') quotes. The main reason to use one over the other is to avoid character escapes within the string literal—for example, if the string literal actually contains double quotes.

By enclosing the string in single quotes, you do not have to escape the embedded double quotes. Consider the following two examples, which are both valid:

```
var quote = "Winston Churchill said:
        \"Never in the field of human conflict was
        so much owed by so many to so few.\""

var quote = 'Winston Churchill said:
         "Never in the field of human conflict was
         so much owed by so many to so few."'
```

Expressions can be embedded within the string literal by using curly braces:

```
var name = "Jim";
// prints My name is Jim
println ( "My name is {name}" );
```

The embedded expression must be a valid JavaFX or Java expression that returns an object. This object will be converted to a string using its `toString()` method. For instance:

```
println ( "Today is {java.util.Date{}}" );

var state ="The state is {
                     if(running) "Running" else "Stopped"}";

println(" The state is {getStateStr()}" );

println("The state is {
             if(checkRunning()) "Running" else "Stopped"}");
```

Also, a string literal may be split across lines:

```
var quote = "Winston Churchill said: "
"\"Never in the field of human conflict was so much owed "
"by so many to so few.\"";
```

In this example, the strings from both lines are concatenated into one string. Only the string literals within the quotes are used and any white space outside of the quotes is ignored.

Unicode characters can be entered within the string literal using `\u` + *the four digit unicode*.

```
var thanks = "dank\u00eb"; // dankë
```

Formatting

Embedded expressions within string literals may contain a formatting code that specifies how the embedded expression should be presented. Consider the following:

```
var totalCountMessage = "The total count is {total}";
```

Now if `total` is an integer, the resulting string will show the decimal number; but if `total` is a Number, the resulting string will show the number formatted according to the local locale.

```
var total = 1000.0;
```

produces:

```
The total count is 1000.0
```

To format an expression, you need a format code within the embedded expression. This is a percent (%) followed by the format codes. The format code is defined in the `java.util.Formatter` class. Please refer to its JavaDoc page for more details (http://java.sun.com/javase/6/docs/api/index.html).

```
println("Total is {%f total}");     // Total is 1000.000000
println("Total is {%.2f total}");   // Total is 1000.00
println("Total is {%5.0f total}");  // Total is  1000
println("Total is {%+5.0f total}"); // Total is +1000
println("Total is {%,5.0f total}"); // Total is 1,000
```

Developer Note: To include a percent (%) character in a string, it needs to be escaped with another percent (%%). For example:

```
println("%%{percentage}"); // prints %25
```

Internationalization

To internationalize a string, you must use the "Translate Key" syntax within the string declaration. To create a translate key, the String assignment starts with `##` (sharp, sharp) combination to indicate that the string is to be translated to the host locale. The `##` combination is before the leading double or single quote. Optionally, a key may be specified within square brackets (`[]`). If a key is not specified, the string itself becomes the key into the locale properties file. For example:

```
var postalCode = ## "Zip Code: ";
var postalCode = ##[postal]"Zip Code: ";
```

In the preceding example, using the first form, the key is `"Zip Code: "`, whereas for the second form, the key is `"postal"`. So how does this work?

By default, the localizer searches for a property file for each unique script name. This is the package name plus script filename with a locale and a file type of `.fxproperties`. So, if your script name is `com.mycompany.MyClass`, the localizer code would look for a property file named `com/mycompany/MyClass_xx.fxproperties` on the classpath, where `xx` is the locale. For example, for English in the United Kingdom, the properties filename would be `com/mycompany/MyClass_en_GB.fxproperties`, whereas French Canadian would be `com/mycompany/MyClass_fr_CA.fxproperties`. If your default locale is just English, the properties file would be `MyClass_en.fxproperties`. The more specific file is searched first, then the least specific file is consulted. For instance, `MyClass_en_GB.fxproperties` is searched for the key and if it is not found, then `MyClass_en.fxproperties` would be searched. If the key cannot be found at all, the string itself is used as the default. Here are some examples:

Example #1:

```
println(##"Thank you");
```

French – MyClass_fr.fxproperties:

```
"Thank you" = "Merci"
```

German – MyClass_de.fxproperties:

```
"Thank you" = "Danke"
```

Japanese – MyClass_ja.fxproperties:

```
"Thank you" = "Arigato"
```

Example #2:

```
println(##[ThankKey] "Thank you");
```

French – MyClass_fr.fxproperties:

```
"ThankKey" = "Merci"
```

German – MyClass_de.fxproperties:

```
"ThankKey" = "Danke"
```

Japanese – MyClass_ja.fxproperties:

```
"ThankKey" = "Arigato"
```

When you use a string with an embedded expression, the literal key contains a `%s`, where the expression is located within the string. For example:

```
println(##"Hello, my name is {firstname}");
```

In this case, the key is "`Hello, my name is %s`". Likewise, if you use more than one expression, the key contains a "`%s`" for each expression:

```
println(##"Hello, my name is {firstname} {lastname}");
```

Now, the key is "`Hello, my name is %s %s`".

This parameter substitution is also used in the translated strings. For example:

French – MyClass_fr.fxproperties:

```
"Hello, my name is %s %s" = "Bonjour, je m'appelle %s %s"
```

Lastly, you can associate another Properties file to the script. This is done using the `javafx.util.StringLocalizer` class. For example:

```
StringLocalizer.associate("com.mycompany.resources.MyResources",
"com.mycompany");
```

Now, all translation lookups for scripts in the `com.mycompany` package will look for the properties file `com/mycompany/resources/MyResources_xx.fxproperties`, instead of using the default that uses the script name. Again, `xx` is replaced with the locale abbreviation codes.

Expressions and Operators

Block Expression

A block expression is a list of statements that may include variable declarations or other expressions within curly braces. If the last statement is an expression, the value of a block expression is the value of that last expression; otherwise, the block expression does not represent a value. Listing 3.6 shows two block expressions. The first expression evaluates to a number represented by the *subtotal* value. The second block expression does not evaluate to any value as the last expression is a `println()` function that is declared as a `Void`.

Listing 3.6 Block Expressions

```
// block expression with a value
var total = {
    var subtotal = 0;
    var ndx = 0;
    while(ndx < 100) {
        subtotal += ndx;
        ndx++;
    };
    subtotal; // last expression
};

//block expression without a value
{
    var total = 0;
    var ndx = 0;
    while(ndx < 100) {
        total += ndx;
        ndx++;
    };
    println("Total is {total}");
}
```

Exception Handling

The `throw` statement is the same as Java and can only throw a class that extends `java.lang.Throwable`.

The `try/catch/finally` expression is the same as Java, but uses the JavaFX syntax:

```
try {
} catch (e:SomeException) {
} finally {
}
```

Operators

Table 3.2 contains a list of the operators used in JavaFX. The priority column indicates the operator evaluation precedence, with higher precedence operators in the first rows. Operators with the same precedence level are evaluated equally. Assignment operators are evaluated right to left, whereas all others are evaluated left to right. Parentheses may be used to alter this default evaluation order.

Table 3.2 Operators

Priority	Operator	Meaning
1	++/-- (Suffixed)	Post-increment/decrement assignment
2	++/-- (Prefixed)	Pre-increment/decrement assignment
	-	Unary minus
	not	Logical complement; inverts value of a Boolean
	sizeof	Size of a sequence
	reverse	Reverse sequence order
	indexof	Index of a sequence element
3	/, *, mod	Arithmetic operators
4	+, -	Arithmetic operators
5	==, !=	Comparison operators (Note: all comparisons are similar to `isEquals()` in Java)
	<, <=, >, >=	Numeric comparison operators
6	instanceof, as	Type operators
7	and	Logical AND
8	or	Logical OR
9	+=, -=, *=, /=	Compound assignment
10	=>, tween	Animation interpolation operators
11	=	Assignment

Conditional Expressions

if/else

`if` is similar to `if` as defined in other languages. First, a condition is evaluated and if true, the expression block is evaluated. Otherwise, if an `else` expression block is provided, that expression block is evaluated.

```
if (date == today) {
      println("Date is today");
}else {
      println("Out of date!!!");
}
```

One important feature of `if/else` is that each expression block may evaluate to an expression that may be assigned to a variable:

```
var outOfDateMessage = if(date==today) "Date is today"
                                  else "Out of Date";
```

Also the expression blocks can be more complex than simple expressions. Listing 3.7 shows a complex assignment using an `if/else` statement to assign the value to `outOfDateMessage`.

Listing 3.7 Complex Assignment Using if/else Expression

```
var outOfDateMessage = if(date==today) {
         var total = 0;
         for(item in items) {
            total += items.price;
         }
         totalPrice += total;
         "Date is today";
    } else {
         errorFlag = true;
         "Out of Date";
    };
```

In the previous example, the last expression in the block, the error message string literal, is the object that is assigned to the variable. This can be any JavaFX Object, including numbers.

Because the `if/else` is an expression block, it can be used with another `if/else` statement. For example:

```
var taxBracket = if(income < 8025.0) 0.10
       else if(income < 32550.0)0.15
       else if (income < 78850.0) 0.25
       else if (income < 164550.0) 0.28
       else 0.33;
```

Looping Expressions

For

`for` loops are used with sequences and allow you to iterate over the members of a sequence.

```
var daysOfWeek : String[] =
                  [ "Sunday", "Monday", "Tuesday" ];
for(day in daysOfWeek) {
      println("{indexof day}). {day}");
}
```

To be similar with traditional `for` loops that iterate over a count, use an integer sequence range defined within square brackets.

```
for( i in [0..100]} {
```

The `for` expression can also return a new sequence. For each iteration, if the expression block executed evaluates to an `Object`, that `Object` is inserted into a new sequence returned by the `for` expression. For example, in the following `for` expression, a new `Text` node is created with each iteration of the day of the week. The overall `for` expression returns a new sequence containing `Text` graphical elements, one for each day of the week.

```
var textNodes: Text[] = for( day in daysOfWeek) {
    Text {content: day };
}
```

Another feature of the `for` expression is that it can do nested loops. Listing 3.8 shows an example of using nested loops.

Listing 3.8 Nested For Loop

```
class Course {
    var title: String;
    var students: String[];
}
var courses = [
    Course {
        title: "Geometry I"
        students: [ "Clarke, "Connors", "Bruno" ]
    },
    Course {
        title: "Geometry II"
        students: [ "Clarke, "Connors",  ]
    },
    Course {
        title: "Algebra I"
        students: [ "Connors", "Bruno" ]
    },
];

for(course in courses, student in course.students) {
   println("Student: {student} is in course {course}");
}
```

This prints out:

```
Student: Clarke is in course Geometry I
Student: Connors is in course Geometry I
Student: Bruno is in course Geometry I
Student: Clarke is in course Geometry II
Student: Connors is in course Geometry II
Student: Connors is in course Algebra I
Student: Bruno is in course Algebra I
```

There may be zero or more secondary loops and they are separated from the previous ones by a comma, and may reference any element from the previous loops.

You can also include a `where` clause on the sequence to limit the iteration to only those elements where the `where` clause evaluates to true:

```
var evenNumbers = for( i in [0..1000] where i mod 2 == 0 ) i;
```

while

The `while` loop works similar to the `while` loop as seen in other languages:

```
var ndx = 0;
while ( ndx < 100) {
    println("{ndx}");
    ndx++;
}
```

Note that unlike the JavaFX `for` loop, the `while` loop does not return any expression, so it cannot be used to create a sequence.

Break/Continue

`break` and `continue` control loop iterations. `break` is used to quit the loop altogether. It causes all the looping to stop from that point. On the other hand, `continue` just causes the current iteration to stop, and the loop resumes with the next iteration. Listing 3.9 demonstrates how these are used.

Listing 3.9 Break/Continue

```
for(student in students) {
   if(student.name == "Jim") {
      foundStudent = student;
      break; // stops the loop altogether,
             //no more students are checked
   }
}

for(book in Books ) {
      if(book.publisher == "Addison Wesley") {
         insert book into bookList;
         continue; // moves on to check next book.
      }
      insert book into otherBookList;
      otherPrice += book.price;
}
```

Type Operators

The `instanceof` operator allows you to test the class type of an object, whereas the `as` operator allows you to cast an object to another class. One way this is useful is to cast a generalized object to a more specific class in order to perform a function from that more specialized class. Of course, the object must inherently be that kind of class, and that is where the `instanceof` operator is useful to test if the object is indeed that kind of class. If you try to cast an object to a class that that object does not inherit from, you will get an exception.

In the following listing, the `printLower()` function will translate a string to lowercase, but for other types of objects, it will just print it as is. First, the generic object is tested to see if it is a `String`. If it is, the object is cast to a `String` using the `as` operator, and then the `String`'s `toLowerCase()` method is used to convert the output to all lowercase. Listing 3.10 illustrates the use of the `instanceof` and `as` operators.

Listing 3.10 Type Operators

```
function printLower(object: Object ) {
    if(object instanceof String) {
        var str = object as String;
        println(str.toLowerCase());
    }else {
        println(object);
    }

}
printLower("Rich Internet Application");
printLower(3.14);
```

Accessing Command-Line Arguments

For a pure script that does not declare exported classes, variables, or functions, the command-line arguments can be retrieved using the `javafx.lang.FX.getArguments():String[]` function. This returns a `Sequence` of `String`s that contains the arguments passed to the script when it started. There is a another version of this for use in other invocations, such as applets, where the arguments are passed using name value pairs, `javafx.lang.FX.getArguments(key:String):String[]`. Similarly, there is a function to get system properties, `javafx.lang.FX.getProperty(key:String):String[]`.

If the script contains any exported classes, variables, or functions, arguments are obtained by defining a special `run` function at the script level.

```
public function run(args:String[] ) {
      for(arg in args) {
            println("{arg}");
      }
}
```

Loose Expressions with Exported Members: Variables, functions, and expressions at the script level (not within a class declaration) are called *loose expressions*. When these variables and functions are private to the script, no specific `run` function is required if the script is executed from the command line. However, if any of these expressions are exported outside of the script using public, public-read, protected, package access, a `run` function is required if the script is to be executed directly. This `run` method encapsulates the exported variables and functions.

Built-in Functions and Variables

There are a set of functions that are automatically available to all JavaFX scripts. These functions are defined in `javafx.lang.Builtins`.

You have already seen one of these, `println()`. `Println()` takes an object argument and prints it out to the console, one line at a time. It is similar to the Java method, `System.out.println()`. Its companion function is `print()`. `Print()` prints out its argument but without a new line. The argument's `toString()` method is invoked to print out a string.

```
println("This is printed on a single line");
print("This is printed without a new line");
```

Another function from `javafx.lang.Builtins` is `isInitialized()`. This method takes a JavaFX object and indicates whether the object has been completely initialized. It is useful in variable triggers to determine the current state of the object during initialization. There may be times that you want to execute some functionality only after the object has passed the initialization stage. For example, Listing 3.11 shows the built-in, `isInitialized()` being used in an `on replace` trigger.

Listing 3.11 isInitialized()

```
public class Test {
    public var status: Number on replace {
        // will not be initialized
        // until status is assigned a value
        if(isInitialized(status)) {

            commenceTest(status);
        }
    }
    public function commenceTest(status:Number) : Void {
        println("commenceTest status = {status}:);
    }
}
```

In this example, when the class, `Test`, is first instantiated, the instance variable, `status`, first takes on the default value of 0.0, and then the `on replace` expression block is evaluated. However, this leaves the status in the `uninitialized` state. Only when a value is assigned to `status`, will the state change to `initialized`. Consider the following:

```
var test = Test{}; // status is uninitialized
test.status = 1; // now status becomes initialized
```

In this case when `Test` is created using the object literal, `Test{}`, status takes on the default value of 0.0; however, it is not `initialized`, so `commenceTest` will not be invoked during object creation. Now when we assign a value to `status`, the state changes to `initialized`, so `commenceTest` is now invoked. Please note that if we had assigned a default value to `status`, even if that value is 0, then `status` immediately is set to `initialized`. The following example demonstrates this.

```
public class Test {
    public var status: Number = 0 on replace {
        // will be initialized immediately.
        if(isInitialized(status)) {
             commenceTest(status);
        }
    }
```

The last built-in function is `isSameObject()`. `isSameObject()` indicates if the two arguments actually are the same instance. This is opposed to the == operator. In JavaFX, the == operator determines whether two objects are considered equal, but that does not mean they are the same instance. The == operator is similar to the Java function `isEquals()`, whereas JavaFX `isSameObject` is similar to the Java == operator. A little confusing if your background is Java!

The built-in variables are `__DIR__` and `__FILE__`. `__FILE__` holds the resource URL string for the containing JavaFX class. `__DIR__` holds the resource URL string for directory that contains the current class. For example,

```
    println("DIR = {__DIR__}");
    println("FILE = {__FILE__}");
    // to locate an image
    var image = Image { url: "{__DIR__}images/foo.jpeg" };
```

The following examples show the output from a directory based classpath versus using a JAR-based class path.

```
Using a Jar file in classpath
$javafx -cp Misc.jar misc.Test


DIR = jar:file:/export/home/jclarke/Documents/
     Book/FX/code/Chapter3/Misc/dist/Misc.jar!/misc/


FILE = jar:file:/export/home/jclarke/Documents/
  Book/FX/code/Chapter3/Misc/dist/Misc.jar!/misc/Test.class
```

continues

```
Using directory classpath
$ javafx -cp . misc.Test

DIR = file:/export/home/jclarke/Documents/Book/
      FX/code/Chapter3/Misc/dist/tmp/misc/


FILE = file:/export/home/jclarke/Documents/Book/
      FX/code/Chapter3/Misc/dist/tmp/misc/Test.class
```

Notice the Trailing Slash on __DIR__: Because the tailing slash already exists on `__DIR__`, ***do not*** add an extra trailing slash when using `__DIR__` to build a path to a resource like an image. `Image{ url: "{__DIR__}image/foo.jpeg"}` is correct.

`Image{ url: "{__DIR__}/image/foo.jpeg"}` ***is wrong***. If you add the trailing slash after `__DIR__`, the image will not be found and you will be scratching your head trying to figure out why not.

Chapter Summary

This chapter covered key concepts in the JavaFX Scripting language. You were shown what constitutes a script and what constitutes a class. You were shown how to declare script and instance variables, how to create and modify sequences, and how to control logic flow.

You now have a basic understanding of the JavaFX Script language syntax and operators. Now, it is time to put this to use. In the following chapters, we will drill down into the key features of JavaFX and show how to leverage the JavaFX Script language to take advantage of those features. In the next chapter, we start our exploration of JavaFX by discussing the data synchronization support in the JavaFX runtime.

Index