



JavaFX 9 by Example

Rich-client applications for any platform

Third Edition

Carl Dea
Gerrit Grunwald
José Pereda, Ph.D
Sean Phillips
Mark Heckler

Apress®

JavaFX 9 by Example

Third Edition



Carl Dea

Gerrit Grunwald

José Pereda, Ph.D

Sean Phillips

Mark Heckler

Apress®

JavaFX 9 by Example

Carl Dea
Pasadena, Maryland, USA

José Pereda, Ph.D
Arroyo de la Encomienda, Spain

Mark Heckler
Godfrey, Illinois, USA

ISBN-13 (pbk): 978-1-4842-1960-7
DOI 10.1007/978-1-4842-1961-4

Gerrit Grunwald
Münster, Nordrhein-Westfalen, Germany

Sean Phillips
Bowie, Maryland, USA

ISBN-13 (electronic): 978-1-4842-1961-4

Library of Congress Control Number: 2017952397

Copyright © 2017 by Carl Dea, Gerrit Grunwald, José Pereda, Sean Phillips and Mark Heckler

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Cover image by Freepik (www.freepik.com)

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Jonathan Gennick
Development Editor: Laura Berendson
Technical Reviewer: Brian Molt
Coordinating Editor: Jill Balzano
Copy Editor: Kezia Endsley

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit <http://www.apress.com/rights-permissions>.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484219607. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

Contents at a Glance

About the Authors.....	xix
About the Technical Reviewer	xxi
Acknowledgments	xxiii
Introduction	xxv
■ Chapter 1: Getting Started	1
■ Chapter 2: JavaFX and Jigsaw	47
■ Chapter 3: JavaFX Fundamentals	69
■ Chapter 4: Lambdas and Properties	103
■ Chapter 5: Layouts and Scene Builder.....	135
■ Chapter 6: User Interface Controls	171
■ Chapter 7: Graphics	227
■ Chapter 8: JavaFX Printing.....	265
■ Chapter 9: Media and JavaFX.....	283
■ Chapter 10: JavaFX on the Web.....	327
■ Chapter 11: JavaFX 3D	367
■ Chapter 12: JavaFX and Arduino	391
■ Chapter 13: JavaFX on Mobile.....	431

■ CONTENTS AT A GLANCE

■ Chapter 14: JavaFX and Gestures	467
■ Chapter 15: Custom UIs	491
■ Chapter 16: Appendix A: References	525
Index.....	551

Contents

About the Authors.....	xix
About the Technical Reviewer	xxi
Acknowledgments	xxiii
Introduction	xxv
■ Chapter 1: Getting Started	1
Downloading Required Software.....	1
Installing the Java 9 Development Kit.....	3
Installing the JDK on Microsoft Windows	3
Installing the JDK on MacOS X	7
Installing the JDK on Linux	11
Setting Environment Variables	14
Setup Windows Environment Variables	16
Setting Up MacOS X/Linux Environment Variables	19
Installing Gradle	22
Installing the NetBeans IDE.....	23
Creating a JavaFX HelloWorld Application.....	29
Using the NetBeans IDE.....	30
Using an Editor and the Terminal (the Command-Line Prompt)	34
Using Gradle on the Command-Line Prompt	38
Walking Through the HelloWorld Source Code	41
JavaFX Scene Graph.....	42
JavaFX Node.....	42

Packaging a JavaFX Application.....	43
Downloading the Book's Source Code.....	44
Summary.....	45
Chapter 2: JavaFX and Jigsaw	47
What Is Project Jigsaw?	48
Benefits	48
Drawbacks.....	49
Java 9 Migration Path.....	49
History.....	52
JAR Hell	52
OSGi.....	53
Maven/Gradle	54
Getting Started	55
What Is the Module Path?.....	56
Module Definition	57
Module Types.....	58
An Example HelloWorld JavaFX 9 Modular Application	63
Create Project Structure	63
Create a Module Definition	63
Create Main Application Code.....	64
Compile Code (Module).....	65
Copy Resources	65
Run Application.....	66
Package Application as JAR	66
Run Application as JAR.....	67
Display Module Description.....	67
Summary.....	68

■ Chapter 3: JavaFX Fundamentals	69
JavaFX Lines	69
Drawing Lines	74
Drawing Shapes	78
Drawing Complex Shapes	79
A Complex Shape Example.....	79
The Cubic Curve	83
The Ice Cream Cone.....	84
The Smile.....	86
The Donut	86
Painting Colors	88
An Example of Color	88
Gradient Color.....	92
Stop Color	92
Linear Gradient	92
Radial Gradient	93
Semitransparent Gradients.....	94
Reflective Cycle Gradients.....	94
Drawing Text.....	95
Changing Text Fonts	97
Applying Text Effects	100
Summary	101
■ Chapter 4: Lambdas and Properties	103
Lambda	103
Lambda Expressions.....	104
Functional Interfaces.....	107
Aggregate Operations.....	108

■ CONTENTS

Default Methods	111
An Example Case: Cats Large and Small	111
Code for the Example	112
Explanation of the Code.....	116
Properties and Binding	117
UI Patterns	117
Properties	117
Types of JavaFX Properties.....	118
JavaFX JavaBean.....	120
Property Change Support	121
Binding	122
Bidirectional Binding	123
High-Level Binding	123
Low-Level Binding.....	124
A Logon Dialog Example.....	124
Login Dialog Source Code.....	126
Explanation of the Code.....	130
Summary.....	134
■ Chapter 5: Layouts and Scene Builder.....	135
Layouts.....	135
HBox	136
VBox.....	140
FlowPane	144
BorderPane	144
GridPane	145
Scene Builder	150
Download and Installing Scene Builder	151
Launching Scene Builder.....	151
A Code Walkthrough	169
Summary.....	170

■ Chapter 6: User Interface Controls	171
Labels	171
Custom Fonts.....	173
Fonts as Icons.....	174
Example: Working with Third-Party Font Packs as Icons.....	174
How It Works.....	181
Buttons	185
Button	185
Check Box.....	186
Hyperlink	187
Radio Button.....	187
Example: Button Fun.....	189
Button Fun Instructions	189
Source Code of ButtonFun.java	190
How It Works.....	196
Menus.....	198
Creating Menus and Menu Items.....	198
Invoking a Selected MenuItem	199
Example: Working with Menus	200
How It Works.....	202
Additional Ways to Select Menus and Menu Items.....	203
The ObservableList Collection Class	204
Working with ListViews	204
Example: Hero Picker.....	205
How It Works.....	208
Working with TableViews	209
What Is a Cell Factory?.....	209
Making Table Cells Editable	210
Example: Bosses and Employees Working with Tables	212

■ CONTENTS

Generating a Background Process	221
Creating a Background Task.....	221
Example: File Copy Progress Dialog (BackgroundProcesses).....	222
How It Works.....	225
Summary.....	226
■ Chapter 7: Graphics	227
Working with Images.....	227
Loading Images	228
Displaying Images	230
A Photo Viewer Example.....	231
Features/Instructions.....	231
UML: Class Diagram.....	233
File Descriptions	234
Source Code	235
Animation	255
What Are Key Values?	255
What Are Keyframes?	255
What Is a Timeline?	256
JavaFX Transition Classes	256
Point-and-Click Game Example.....	257
Source Code	257
How It Works.....	261
Compound Transitions	261
PhotoViewer2 Example.....	262
Summary.....	264
■ Chapter 8: JavaFX Printing.....	265
JavaFX Printing	265
JavaFX Print APIs.....	268

Printer and PrinterJob	268
Query Printer Attributes	270
Configuring a Print Job	272
Printing a Web Page	274
Example WebDocPrinter Application	275
Source Code	277
How Does It Work?	281
Summary	282
■ Chapter 9: Media and JavaFX	283
Media Events	283
Playing Audio	285
An MP3 Player Example	285
MP3 Audio Player Source Code	287
How It Works	301
Playing Video	314
MPEG-4	314
VP6 .flv	314
A Video Player Example	315
Video Player Source Code	316
How It Works	320
Simulating Closed Captioning: Marking a Position in a Video Media	323
Closed Captioning Video Example	323
How It Works	325
Summary	325
■ Chapter 10: JavaFX on the Web	327
JavaFX Web and HTTP2 APIs	328
Web Engine	330
WebEngine's load() Method	330
WebEngine's loadContent() Method	331

CONTENTS

HTML DOM Content	331
Obtaining an org.w3c.dom.Document (DOM) Object	331
Using Raw XML Content as a String	332
The JavaScript Bridge	332
Communicating from Java to JavaScript.....	333
Communicating from JavaScript to Java.....	333
Java 9 Module jdk.incubator.httpclient.....	335
Making RESTful Requests.....	339
The HTTP GET Request	340
HTTP POST Request.....	341
WebSockets.....	342
Viewing HTML5 Content (WebView).....	345
Example: An HTML5 Analog Clock	345
Analog Clock Source Code.....	346
How It Works.....	349
Inkscape and SVG.....	349
WebEvents	350
Weather Widget Example.....	351
One-Liner: Reading an Input Stream into a String.....	353
Source Code	354
How It Works.....	363
Enhancements.....	364
Summary.....	365
■ Chapter 11: JavaFX 3D	367
Basic 3D Scenes in JavaFX	367
A Very Basic 3D Scene Example	367
Primitives	369
Adding a Primitive Example	369
Simple Translate and Rotate Example	371
Multiple Primitive Transformation Example	372
All Together Now: Grouped Primitives	373

Interacting with Your Scene	374
Primitive Picking for Primitives	375
First Person Movement Using the Keyboard.....	376
First Person Camera Movement Using the Mouse.....	377
Beyond the Basics.....	379
Custom 3D Objects Using the TriangleMesh Class	380
“Winding” and Wuthering.....	380
MeshViews and DrawMode	383
Roll Camera!.....	388
Hit the Lights	389
Summary.....	390
■ Chapter 12: JavaFX and Arduino	391
The Arduino Board.....	391
Programming the Arduino	393
Arduino Web Editor	394
Arduino IDE.....	396
Windows	396
MacOS X or Linux	398
Running the IDE	398
The Blink Example.....	400
Orientation Visualizer Example.....	401
How It Works.....	406
Serial Reading	406
Java Simple Serial Connector.....	406
JavaFX, the Charting API, and Orientation	406
Creating the Module Project	407
Serial Communications.....	409
How It Works.....	412
Testing Serial Comms.....	414
The JavaFX Charts API.....	414
Building and Running the Project	420

■ CONTENTS

How It Works.....	421
Adding More Functionality.....	425
Building and Running the Project.....	427
How It Works.....	428
More Examples.....	430
Summary.....	430
■ Chapter 13: JavaFX on Mobile.....	431
JavaFXPorts: The Port to Mobile.....	431
JavaFXPorts Under the Hood	431
Getting Started with JavaFXPorts	432
Hello Mobile World Example.....	433
How Does It Work?	435
Submitting the App to the Stores.....	437
Gluon Mobile	438
The Gluon IDE Plug-Ins	438
Charm Glisten	439
License	441
Example: The BasketStats App.....	441
Creating the Project.....	441
Adding the Model.....	447
Adding the Service	450
Modifying the Main View	453
Modifying the Board View.....	457
Deploy to Mobile.....	464
More Examples.....	465
Summary.....	465
■ Chapter 14: JavaFX and Gestures.....	467
Recognizing Gestures in Your Application	467
Example: Animating Shapes Along a Path Using Touch Events	469
How Does It Work?	472

Touching, Rotating, and Zooming in 3D	473
The Leap Motion Controller	476
How It Works.....	477
Getting Started with the Leap SDK	478
Adding the Leap SDK to a JavaFX Project	479
The Hands Tracking Example.....	479
The LeapListener Class.....	480
The 3D Model Classes	483
The Application Class	485
Building and Running the Project	487
More Examples.....	489
Summary.....	489
■ Chapter 15: Custom UIs	491
Theming	491
Native Look and Feels	493
Web and Mobile Look and Feels.....	495
Applying the JavaFX CSS Theme	497
An Example of Switching Themes	500
JavaFX CSS Styling	505
What Are Selectors?	506
How to Define -fx- Based Styling Properties (Rules).....	511
Obeying the JavaFX CSS Rules.....	512
Custom Controls	513
The LED Custom Control.....	514
Structure of the LED Custom Control Example Code	515
The Properties of the LED Control.....	516
The Initialization Code of the LED Control.....	519
Other Ways to Create a Custom Control.....	523
Summary.....	523

■ Chapter 16: Appendix A: References	525
Java 9 SDK	525
Java 9 API Documentation.....	525
Java 9 Features.....	525
Java 9 Jigsaw	526
IDEs	526
Deploying Applications	526
JavaFX 2D Shapes.....	527
JavaFX Color.....	527
JavaFX 2.x Builder Classes	527
JavaFX Printing	527
Project Lambda	528
Nashorn.....	529
Properties and Bindings	529
Layouts.....	530
JavaFX Tools.....	530
Enterprise GUI Frameworks.....	531
Domain-Specific Languages	532
Custom UIs	532
Operating System Style Guidelines	535
JavaFX Media	535
JavaFX on the Web	536
JavaFX 3D	537
JavaFX Gaming.....	538
Java IoT and JavaFX Embedded.....	539
Software and Device Manufacturers.....	540
JavaFX Communities.....	540

Applications.....	541
Java/JavaFX Books and Magazines	543
Author Blogs.....	544
Tutorials, Courses, Consulting Firms, and Demos	544
Tools, Applications, and Libraries	545
Videos and Presentations on JavaFX	547
Index.....	551

About the Authors



Carl Dea is a principal software engineer. He has been developing software for over 20 years, for many clients from Fortune 500 companies to nonprofit organizations. He has written software ranging from mission-critical applications to ecommerce-based Web applications. His passion for software development started when his middle school science teacher showed him a TRS-80 computer. Carl has been using Java since the very beginning, and he has been a huge JavaFX enthusiast since the early days, when it was its own language called JavaFX script. His current software development interests are UI/UX, game programming, data visualizations, embedded systems, smartphones, AI, and robotics. When Carl is not working, he and his wife enjoy canoeing and day trips to the beach. Carl and his wife are proud parents of their younger daughter who attends Salisbury University. Carl and his wife are also proud of their older daughter, who is a high school teacher in Anne Arundel County Maryland. Carl loves to code socially at <http://github.com/carldea>. He blogs at <http://carlfx.wordpress.com/>. Carl is also connected to LinkedIn at <http://www.linkedin.com/in/carldea>, and his Twitter handle is @carldea. Carl lives in Pasadena, Maryland, USA.



Gerrit Grunwald is a software engineer with more than 10 years of experience in software development. He has been involved in development of Java desktop applications and controls. His current interests include JavaFX, HTML5, and Swing, especially development of custom controls in these technologies. Gerrit is also interested in Java-driven embedded technologies like JavaSE Embedded on Raspberry Pi, i.MX6, BeagleBone Black, CubieBoard2, and similar devices. He is a true believer in open source software and has participated in popular projects like JFXtras.org as well as his own projects (Enzo, SteelSeries Swing, and SteelSeries Canvas). Gerrit is an active member of the Java community, where he founded and leads the Java User Group Münster (Germany), co-leads the JavaFX and IoT community at Oracle, and is a JavaOne RockStar and Java Champion. He is also a speaker at conferences and user groups internationally.

■ ABOUT THE AUTHORS



José Pereda, Ph.D in Structural Engineering, works as a software engineer at Gluon. Being on Java since 1999, he is a JavaFX advocate, developing JavaFX applications for mobile platforms and embedded platforms. He also works on open source projects (JFXtras, FXyz, <https://github.com/jperedadnr>), co-authoring a JavaFX book (JavaFX 8 Introduction by Example), blogging (<http://jperedadnr.blogspot.com.es/>), tweeting (@JPeredaDnr), and speaking at conferences (such as JavaOne, JAX, Jfokus, JavaLand, and Coding Serbia). José lives with his wife and four kids in Valladolid, Spain.



Sean Phillips is a Java software architect currently working on ground systems for Space Science missions at the NASA Goddard Space Flight Center. He has developed embedded systems, along with modeling simulation, and visualization software for projects from commercial and military RADAR, heavy manufacturing, Battlespace Awareness, and Orbital Flight Dynamics. Sean is the lead developer of F(X)yz (<http://birdasaur.github.io/FXyz/>), a free third-party library for JavaFX 3D components and data visualization tools. Sean enjoys sharing his experiments through blog entries on various sites, including Java.net (<https://weblogs.java.net/blogs/sean.mi.phillips>), The NetBeans DZone (<http://netbeans.dzone.com/users/seanmiphillips>), and personal media (Twitter @SeanMiPhillips).

Sean lives in Bowie, Maryland, USA, with his very supportive wife Zulma and sons Sebastian and Sean Alexander.



Mark Heckler is a Principal Technologist & Developer Advocate at Pivotal, a conference speaker, and published author focusing on software development for the Internet of Things and the cloud. He has worked with key players in the manufacturing, retail, medical, scientific, telecom, and financial industries, and with various public sector organizations to develop and deliver critical capabilities on time and on budget. Mark is an open source contributor, author/curator of a developer-focused blog, and on Twitter (@MkHeck). Mark lives with his very understanding wife, three kids, and dog in St. Louis, Missouri, USA.

About the Technical Reviewer



Brian Molt has been a software developer at a manufacturing company in Nebraska for close to 14 years. During his time there, he has programmed with multiple languages on a variety of platforms, including RPGLE on an AS/400, web developing with ASP.NET, desktop application development with JavaFX, and recently REST services with JAX-RS.

Acknowledgments

I would like to thank my amazing wife, Tracey, and my daughters, Caitlin and Gillian, for their loving support and sacrifices. A special thanks to my daughter Caitlin, who helped with illustrations and brainstorming fun examples for the first edition. A big thanks to Jim Weaver for being a great mentor and friend. I would also thank Josh Juneau for his advice and guidance throughout this journey. A big thank you to Brian Molt, for providing excellent feedback and, most of all, for testing my code examples. Thanks also to David Coffin for tech reviewing the second edition. Hats off to my co-authors Gerrit Grunwald, José Pereda, Sean Phillips, and Mark Heckler, for their amazing talents. My co-authors are truly superheroes who came to my rescue to make this book possible.

A huge shout-out to the wonderful people at Apress for their professionalism and support. A special thanks to Jonathan Gennick for believing in me and whipping me into shape again and again. Thanks to Jill Balzano for keeping us on track and getting us across the finish line.

Thanks to all who follow me on Twitter, especially the topics (hash tags) related to JavaFX, UI/UX, and IoT. Also, thanks to the authors (Jim Weaver, Weiqi Gao, Stephen Chin, Dean Iverson, and Johan Vos) of the *Pro JavaFX* book for allowing me to tech review chapters in the first edition. Thanks also to Stephen Chin and Keith Combs for heading up the awesome JavaFX User Group. I want to also thank Rajmahendra Hegde for helping me with the eWidgetFX open source framework effort. Thanks to Hendrik Ebbers for always wanting to team up with me on talks and fun projects at past JavaOne conferences. Most importantly, thanks to all the past JavaFX 1.x book authors that helped inspire me.

A massive thank you to the JavaFX community at large, which there are way too many individuals to name. Lastly, I want to give a big kudos and acknowledgment to the people at Oracle and ex-employees who helped me (directly or indirectly) as JavaFX 2.x, 8, and 9 were being released: Nandini Ramani, Jonathan Giles, Jasper Potts, Richard Bair, Angela Caicedo, Stuart Marks, John Yoon, David Grieve, Michael Heinrichs, David DeHaven, Nicolas Lorain, Kevin Rushforth, Sheila Cepero, Gail Chappell, Cindy Castillo, Scott Hommel, Joni Gordon, Alexander Kouznetsov, Irina Fedortsova, Dmitry Kostovarov, Alla Redko, Nancy Hildebrandt, and all the Java, JavaFX, and NetBeans teams involved.

Whether, then, you eat or drink or whatever you do, do all to the glory of God (1 Corinthians 10:31, NASB).

—Carl Dea

Introduction

Welcome to *JavaFX 9 by Example*.

What is JavaFX?

JavaFX is Java's next-generation graphical user interface (GUI) toolkit that allows developers to rapidly build rich cross-platform applications. Built from the ground up, JavaFX takes advantage of modern GPUs through hardware-accelerated graphics while providing well-designed programming interfaces enabling developers to combine graphics, animation, and UI controls. The new JavaFX 9 is a pure Java language application programming interface (API). The goal of JavaFX is to be used across many types of devices, such as embedded devices, smartphones, TVs, tablet computers, and desktops.

Nandini Ramani, former VP of Development at Oracle, plainly states the intended direction of JavaFX as a platform in the following excerpt from the screencast *Introducing JavaFX*:

The industry is moving toward multi-core/multi-threading platforms with GPUs. JavaFX leverages these attributes to improve execution efficiency and UI design flexibility. Our initial goal is to give architects and developers of enterprise applications a set of tools and APIs to help them build better data-driven business applications.

—Nandini Ramani, former Oracle Corp. VP of Development, Java Client Platform

Before the creation of JavaFX, the development of rich client-side applications involved the gathering of many separate libraries and APIs to achieve highly functional applications. These separate libraries include media, UI controls, Web, 3D, and 2D APIs. Because integrating these APIs together can be rather difficult, the talented engineers at Oracle created a new set of JavaFX libraries that roll up all the same capabilities under one roof. JavaFX is the “Swiss Army Knife” of GUI toolkits (Figure FM-1). JavaFX 9 is a pure Java (language) API that allows developers to leverage existing Java libraries and tools.



Figure FM-1. JavaFX

Depending on who you talk to, you are likely to encounter different definitions of “user experience” (or in the UI world, UX). But one fact remains—the users will always demand better content and increased usability from GUI applications. In light of this fact, developers and designers often work together to craft applications to fulfill this demand. JavaFX provides a toolkit that helps both the developer and designer (in some cases, the same person) create functional, yet aesthetically pleasing, applications. Another thing to acknowledge, when developing a game, media player, or the usual enterprise application, is that JavaFX will not only assist in developing richer UIs, but you’ll also find that the APIs are extremely well-designed to greatly improve developer productivity (we’re all about the user of the API’s perspective).

Although this book doesn’t go through an exhaustive study of all of JavaFX 8 and 9’s capabilities, you will find common use cases that can help you build richer applications. Hopefully, this book will lead you in the right direction by providing practical, real-world examples.

Some History

In 2005 Sun Microsystems acquired the company SeeBeyond, where a software engineer named Chris Oliver created a graphics-rich scripting language known as F3 (Form Follows Function). F3 was later unveiled by Sun Microsystems at the 2007 JavaOne conference as JavaFX. On April 20, 2009, Oracle Corporation announced the acquisition of Sun Microsystems, making Oracle the new steward of JavaFX.

At JavaOne 2010, Oracle announced the JavaFX roadmap, which included its plans to phase out the JavaFX scripting language and re-create the JavaFX platform for the Java platform as Java-based APIs. As promised based on the 2010 roadmap, JavaFX 2.0 SDK was released at JavaOne, in October 2011. In addition to the release of JavaFX 2.0, Oracle announced its commitment to take steps to open source JavaFX, thus allowing the community to help move the platform forward. Open sourcing JavaFX will increase its adoption, enable a quicker turnaround time on bug fixes, and generate new enhancements.

Between JavaFX 2.1 and 2.2, the number of new features grew rapidly. Table FM-1 shows the numerous features included between versions 2.1 and 2.2. JavaFX 2.1 was the official release of the Java SDK on a Mac OS. JavaFX 2.2 was the official release of the Java SDK on the Linux operating system.

The Java 8 release was announced March 18, 2014. Java 8 has many new APIs and language enhancements, which includes lambdas, Stream API, Nashorn JavaScript engine, and JavaFX APIs. Relating to JavaFX 8, the following features include 3D graphics, rich text support, and printing APIs.

Java 9 is expected to be released some time in September of 2017. Java 9 has so many features, but the most important feature is modularization, better known as *Project Jigsaw*.

What You Will Learn in This Book

In this book, you will be learning JavaFX 9 capabilities by following practical examples. These examples will, in turn, provide you with the knowledge needed to create your own rich client applications. Following Java’s mantra, “Write once, run anywhere,” JavaFX also preserves this same sentiment. Because JavaFX 9 is written entirely in the language of Java, you will feel right at home.

Most of the examples can be compiled and run under Java 8. However, some of them are designed to take advantage of Java 9’s language enhancements. While working through this book with JavaFX 8 and 9, you will see that the JavaFX APIs and language enhancements will help you to become a more productive developer. Having said this, we encourage you to explore all the new Java 8 capabilities.

This book covers JavaFX 8 and 9’s fundamentals, modules, lambdas, properties, layouts, UI controls, printer, animation, custom UIs, charts, media, web, 3D, Arduino, touch events, and gestures.

Who This Book Is For

If you are a Java developer looking to take your client-side applications to the next level, you will find this book your guide to help you begin creating usable and aesthetically pleasing user interfaces. Also, if you are an experienced Java Swing, Flash/Flex, SWT, or web developer who wants to learn how to create high-performance rich client-side applications, then this is the book for you.

How This Book Is Structured

This book is arranged in a natural progression from beginner to intermediate and advanced level concepts. For the Java developer, none of the concepts mentioned in this book should be extremely difficult to figure out. This book is purely example-based and discusses major concepts before demonstrating applications. For each project, after showing the output from executing the code, we'll provide a detailed explanation walking through the code. Each example can be easily adapted to meet your own needs when developing a game, media player, or your usual enterprise application. The more experienced a Java UI developer you are, the more freedom you'll have to jump around to different chapters and examples throughout the book. However, any Java developer can progress through the book and learn the skills needed to enhance their everyday GUI applications.

CHAPTER 1



Getting Started

To get started using this book to learn JavaFX 9, you will want to set up your development environment to compile and run many of its examples. In this chapter you will learn how to install the required software, such as the Java development kit (JDK) and the NetBeans integrated development environment (IDE). After installing the required software, you will begin by creating a traditional JavaFX HelloWorld example. Once you are comfortable with the development environment, I will walk through the JavaFX HelloWorld source code. Finally, you will get a glimpse of how to package your application as a standalone application to be launched and distributed.

I realize many developers are partial to their IDEs and editors, so I've also provided three ways to compile and run the HelloWorld example. In the first way to compile and run code you will be using the NetBeans IDE, in the second way you will learn how to use the command line (terminal window), and thirdly you will be using a build tool called *Gradle* to compile and launch JavaFX applications. The second and third strategies are for those who are not fond of fancy IDEs. If you are more comfortable with the *Eclipse* and *IntelliJ* IDE, you will be happy to know that Gradle can generate the respective project files that will allow your projects to be easily loaded into them.

If you are already familiar with the installation of the Java Development Kit (JDK), Gradle, and the NetBeans IDE, you can skip to Chapter 2, which covers Java 9 Jigsaw. Let's get started!

Downloading Required Software

When using this book, you'll find that many examples may still be written using the Java 8 JDK; however Chapter 2 and a few other chapters will get into Java 9-specific concepts, making the Java 9 JDK a requirement. One of the primary changes to the Java platform is modularity (Jigsaw). Modularity will truly transform your client-side applications. You are probably wondering exactly, "how is it transformed?" Well, imagine building a Java application that occupies a smaller footprint in memory and on disk space. This means users will experience faster load times and a better user experience!

Having said this, go ahead and download Java 9 Java Development Kit (JDK) or later. In this chapter, you will see the steps on installing the JDK on Windows, MacOS X, and Linux. You can download the Java 9 JDK from the following URL:

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Another tool that we have not mentioned in prior editions is *Gradle*. Gradle has become an industry standard in building projects and artifacts in the real world. Later, I will walk you through how to install Gradle, but for now head over to gradle.org and look for the download link. At the download link you will be presented with two binary distributions. One is just a binary distribution and the other is a full distribution that includes the source code and documentation. The binary distribution is required to build applications, so you'll want to download it. Go to the following URL to download Gradle's Build tool:

<https://gradle.org/gradle-download>

Installing an integrated development environment (IDE) is optional for the book; however if you choose to use an IDE you will be able get all the benefits of code highlighting, code completion, incremental debugging, refactoring, and many modern developer conveniences. The top three IDEs used in the wild are NetBeans, IntelliJ, and Eclipse. Although this book walks you through installing the NetBeans IDE, it doesn't mean we are partial to NetBeans or any other IDEs. I've chosen NetBeans because release schedules are usually on par with the major JDK releases and it's quite mature, especially when it comes to tooling and JavaFX application development. NetBeans has numerous JavaFX demo projects (example templates) built-in and the Scene Builder tool. Scene Builder is a graphical tool used to build JavaFX UIs visually.

Another pro tip is learning about Gradle *tasks*, which are capable of generating Java projects for either IntelliJ or Eclipse IDE with a single command. To learn more about Gradle tasks that can automatically set up IntelliJ and Eclipse projects, visit the following links:

- IntelliJ: https://docs.gradle.org/current/userguide/idea_plugin.html
- Eclipse: https://docs.gradle.org/current/userguide/eclipse_plugin.html

The following are URLs to download your favorite IDE:

- NetBeans: <https://netbeans.org/downloads>
- IntelliJ: <https://www.jetbrains.com/idea/download>
- Eclipse: <https://eclipse.org/downloads>

Currently, JavaFX 9 from Oracle corp. runs on the following operating systems:

- Windows OS (Vista, 7, 8, 10) 32- and 64-bit
- MacOS X (64-bit)
- Ubuntu Linux 12.x - 13.x (32- and- 64-bit)

To see a detailed listing of all the supported operating systems, visit the following link:

<http://jdk.java.net/9/supported>

While you may not see your supported operating system or hardware, you will be happy to know about the OpenJDK and OpenJFX projects, which allow you to build Java and JavaFX yourself! To learn more about OpenJDK or OpenJFX, visit the following URLs:

- OpenJDK: <http://openjdk.java.net>
- OpenJFX: <https://wiki.openjdk.java.net/display/OpenJFX/Main>

After downloading the appropriate software versions for your operating system, you will install the Java 9 JDK, Gradle, and/or your chosen IDE.

Installing the Java 9 Development Kit

Once you have downloaded the correct version of Java for your particular operating system, follow the steps outlined in this section to install the Java 9 JDK. I assume you've downloaded Oracle's Java 9 JDK, but if not, go to following location:

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

In the following sections, you will follow steps to install Java 9 on the three popular operating systems: Microsoft Windows, MacOS X, and Linux.

Installing the JDK on Microsoft Windows

The following steps use the Java 9 JDK 64-bit version for the Windows 10 operating system. If your Windows operating system is different, refer to the following link for further details.

<http://www.oracle.com/technetwork/java/javase/overview/index.html>

The following are steps to install the Java 9 JDK on the Windows OS:

1. Install the Java 9 JDK by right-clicking the installer's executable (`jdk-9-windows-x64.exe`) to run as an administrator. Typically, other operating systems will pop up a similar warning dialog box to alert you of the risks of installing or running binary files. Click the Run button to proceed.
2. Begin the installation process by clicking on the Next button, as shown in Figure 1-1.

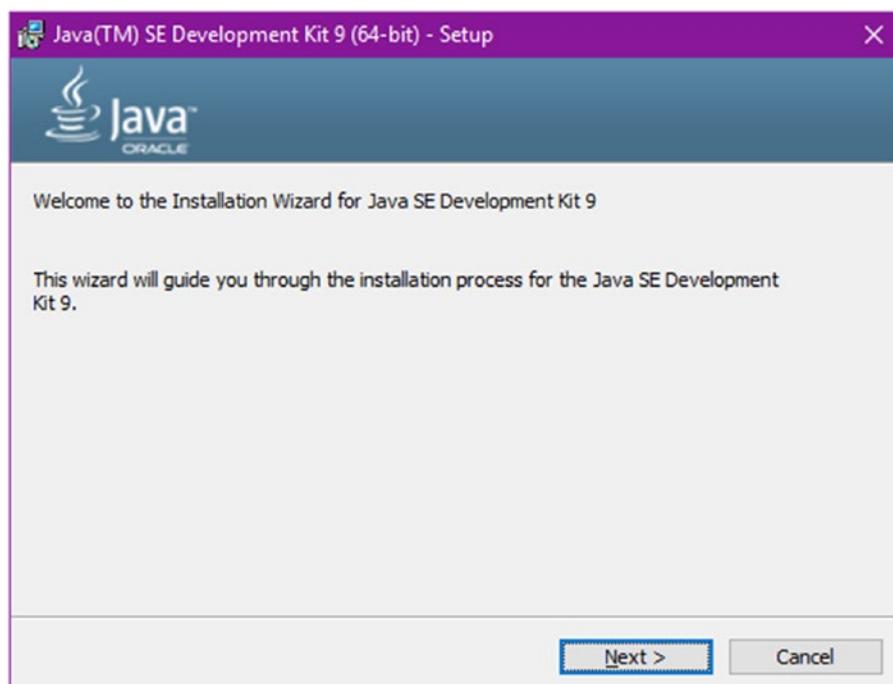


Figure 1-1. Java SE Development Kit 9 installation process

3. Next, you'll be presented with the Custom Setup window allowing you to select optional features shown in Figure 1-2. Here, you'll just accept the default selections and click the Next button to continue.

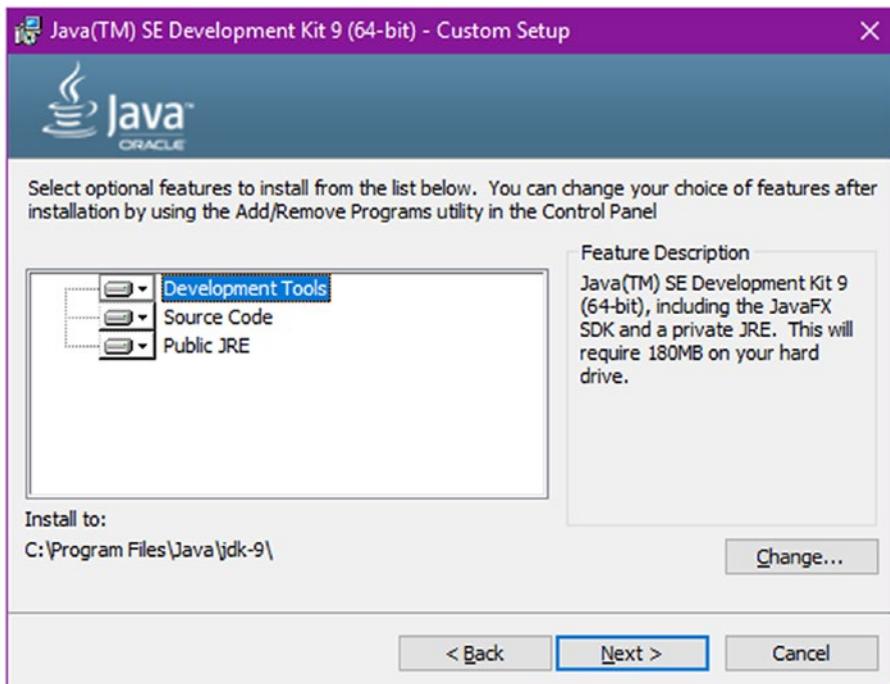


Figure 1-2. Java SE Development Kit 9 optional features

- Afterward, the installation process will display a progress bar, as shown in Figure 1-3.

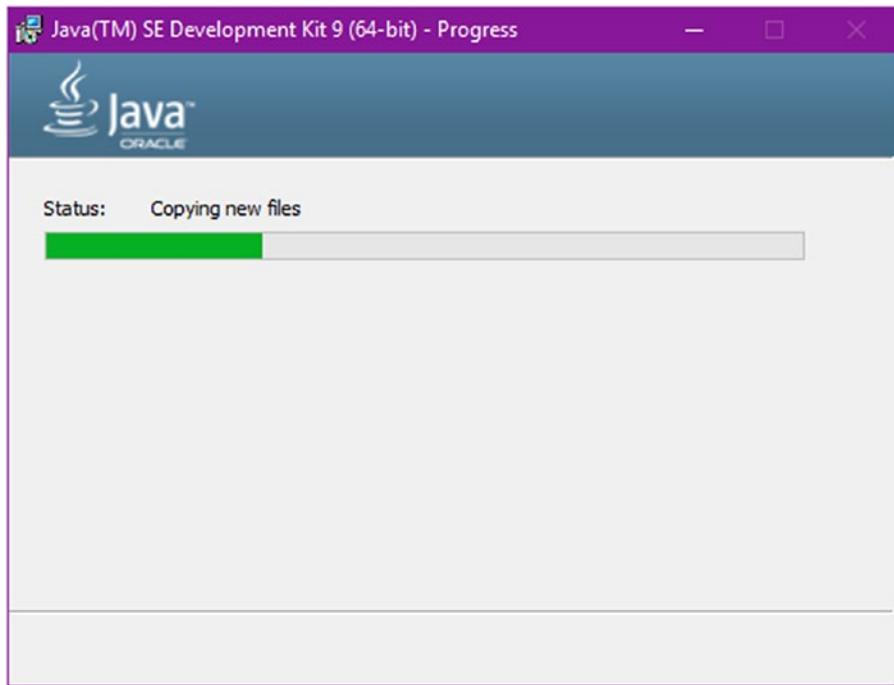


Figure 1-3. Java SE Development Kit 9 installation in progress

5. Next, a popup dialog window will ask if you want to change the installation directory of the Java runtime. Just click the Next button and accept the default directory, as shown in Figure 1-4.

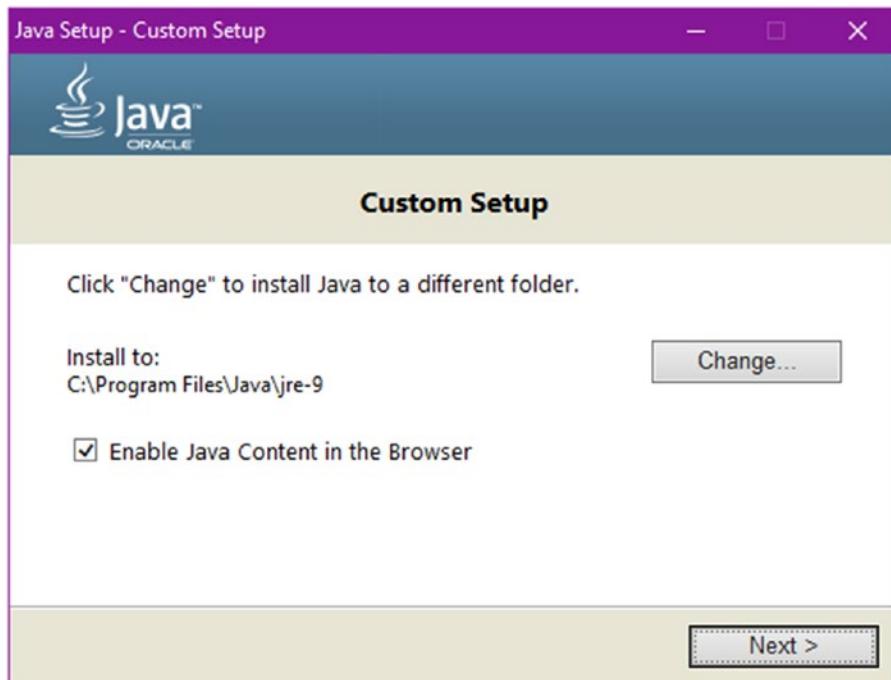


Figure 1-4. The Java 9 runtime destination directory

- To complete the installation of the Java 9 SE Development Kit, click the Close button to exit, as shown in Figure 1-5.

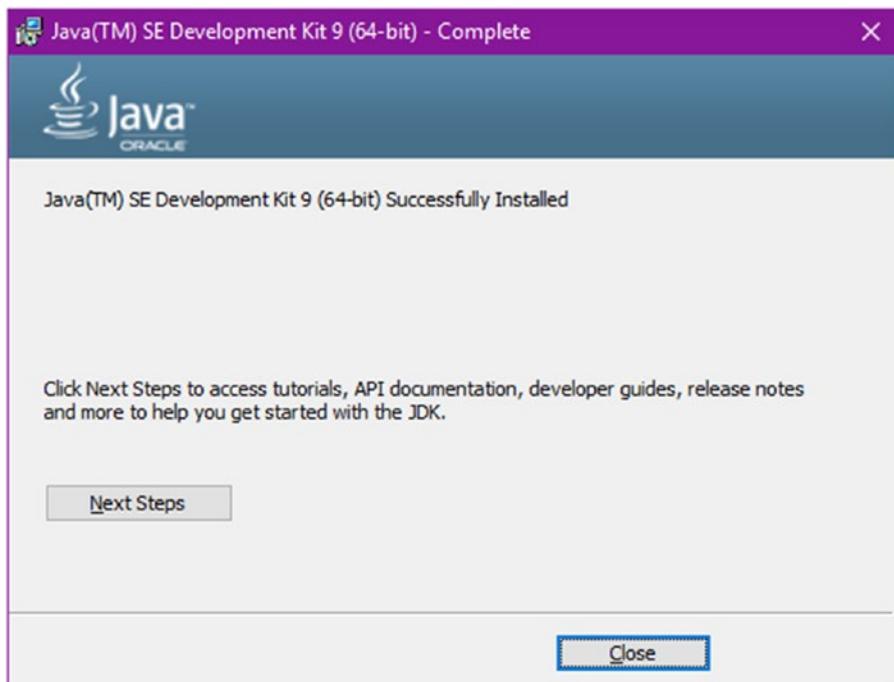


Figure 1-5. The last part of the installation prompting that the Java SE Development Kit 9 was successfully installed

Installing the JDK on MacOS X

The following steps use the Java 9 JDK 64-bit version for the MacOS X (El Capitan) operating system. It assumes you've already downloaded the JDK and it's called something similar to the following:

`jdk-9-ea-bin-b88-macosx-x86_64-21_oct_2015.dmg`

Use the following steps to install the Java 9 JDK on the MacOS X operating system:

- After launching the `dmg` file, you'll be presented the following dialog box as shown in Figure 1-6. Next, double-click on the icon to begin the install process.



Figure 1-6. The Java 9 JDK installer on the MacOS X operating system

2. Next, to begin the install process shown in Figure 1-7, click on Continue. For the rest of the installation process, accept the defaults.



Figure 1-7. The Java 9 JDK installer introduction for the MacOS X operating system

3. After clicking on the Continue button, the dialog will mention the amount of disk space the JDK will take up on your computer, as shown in Figure 1-8. Click on the Install button to proceed.

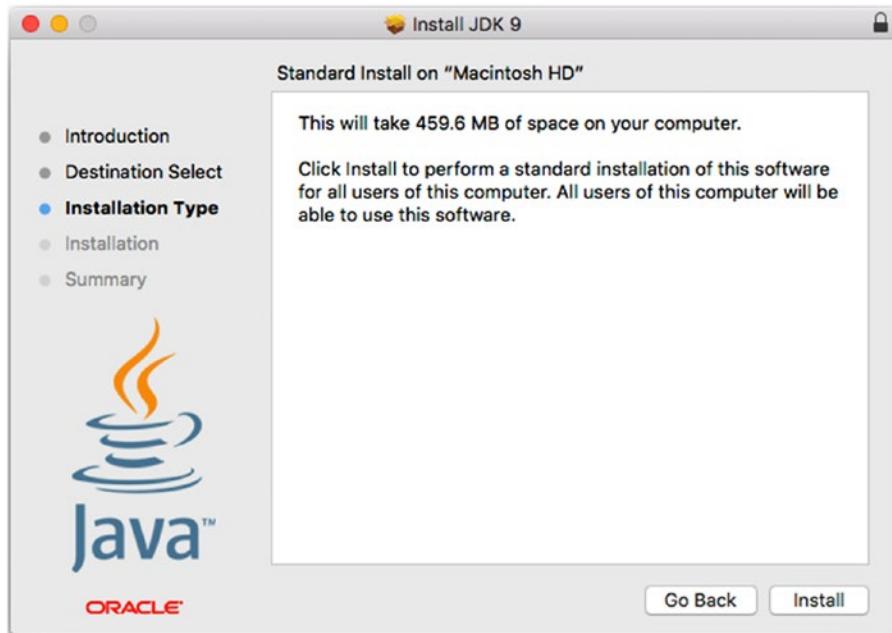


Figure 1-8. The prompt showing how much space the JDK will use

4. For security purposes, your MacOS X will prompt you with a password to allow the installer to continue, as shown in Figure 1-9. This assumes you have admin privileges to install software. After entering your password, click the Install Software button to kick off the process.

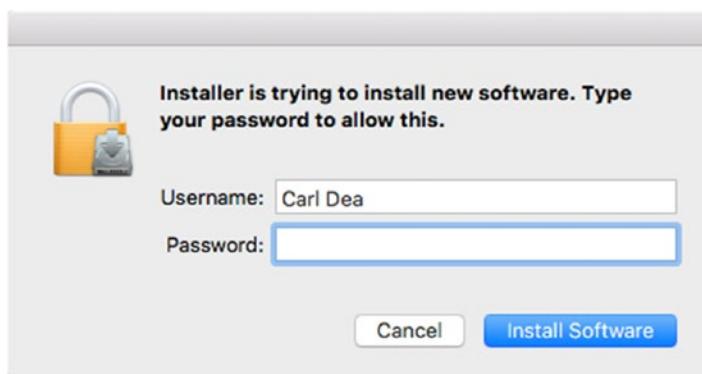


Figure 1-9. The permission window prompting for the username and password that will allow the JDK software to be install on the local machine

5. The installation process will take a few seconds, as shown in Figure 1-10.

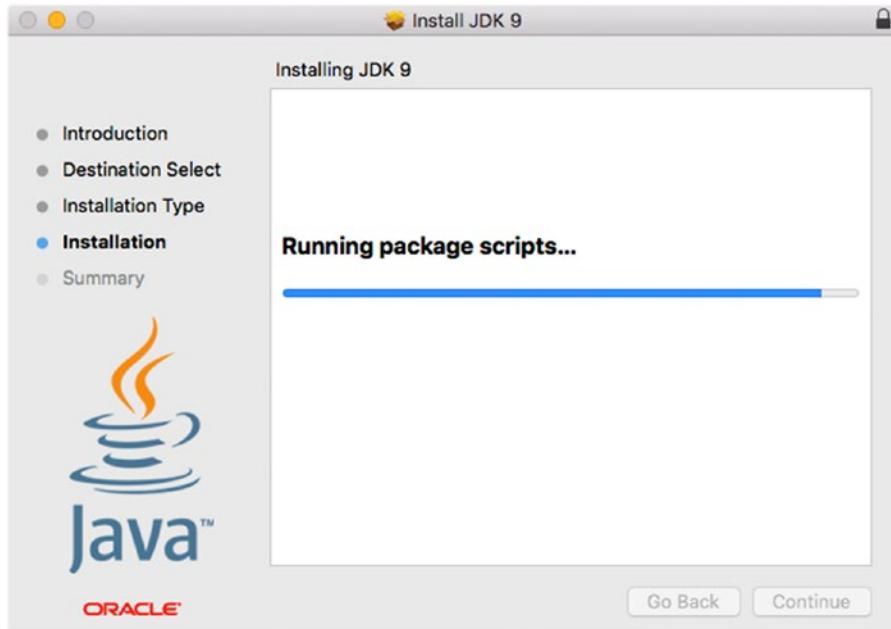


Figure 1-10. The installation of the JDK in progress

6. Lastly, you will click on the Close button, as shown in Figure 1-11. This completes the JDK installation on the MacOS operating system.

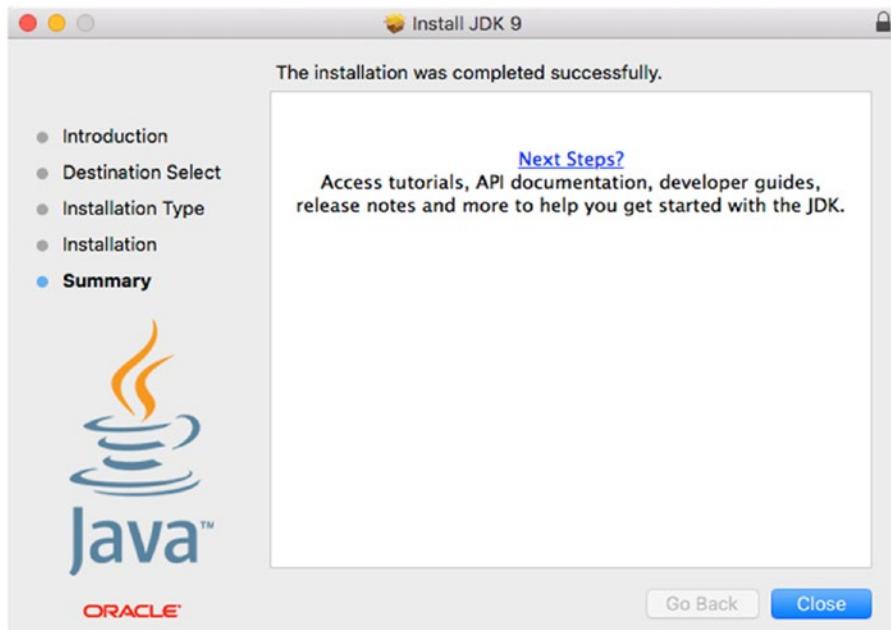


Figure 1-11. The installation of the JDK is complete

After finishing the installation of the JDK, you'll learn later in this chapter how to set up your environment variables. Next are instructions for installing JDK 9 on Linux-based operating systems.

Installing the JDK on Linux

To begin, you should determine if your Linux system is a 32-bit or 64-bit architecture by typing in the following in a terminal:

```
$ uname -m
```

The output on a 64 bit Linux machine is as follows:

```
x86_64
```

The output on a 32 bit Linux machine is as follows:

```
i686
```

Assuming you've downloaded the Java JDK at the following:

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

For users in the Linux world, you will need to download a file named something like:

- jdk-9-bin-b88-linux-i586-21_oct_2015.tar.gz
- jdk-9-bin-b88-linux-i586-21_oct_2015.rpm.

Locate the downloaded file (*.gz or *.rpm) and then go to your terminal window to begin the installation process. The instructions for the installation process are in the next section.

Note As of the writing of this book, the Java 9 JDK used was an EA (early edition) release and therefore, the JAVA_HOME directory name is subject to change. Visit Java 9 JDK's installation instructions for further details here: <https://docs.oracle.com/javase/9/install/installation-jdk-and-jre-linux-platforms.htm>.

Fedora, CentOS, Oracle Linux, or Red Hat Enterprise Linux OSes

After downloading the gz file, you will need to decompress it. On Fedora, CentOS, Oracle Linux, or Red Hat Enterprise Linux OSes, type the following commands to install the JDK.

If a user isn't registered in the sudoers file, you will need to su as root to perform the following commands.

```
$ mv ~/Downloads/jdk-9-linux-x64.tar.gz /tmp  
$ cd /tmp  
$ tar zxvf /tmp/jdk-9-linux-x64.tar.gz  
$ sudo mkdir /opt/jdk/  
$ sudo mv jdk-9 /opt/jdk
```

Use the following command if you downloaded the rpm version of the Java 9 JDK:

```
$ sudo rpm -ivh jdk-9-linux-x64.rpm
```

Red Hat Alternatives

After installing JDK 9 your system, you will likely encounter other versions of Java previously installed. Red Hat/CentOS-based operating systems have a way to switch between JDKs and JREs. Since you have installed the JDK 9 in the /opt/jdk directory, you will want alternatives to register the new JDK 9 as the default Java.

To begin let's register the newly installed Java executable in alternatives. You will want to enter the following commands. These commands will update any indexes, register Java with a priority 100, and display all registered Java runtimes that are installed.

```
$ sudo update
$ sudo alternatives --install /usr/bin/java java /opt/jdk/jdk-9/bin/java 100
```

Next, you need to set the default Java version that you just added.

```
$ sudo alternatives --config java
```

There are 3 programs which provide 'java'.

Selection	Command
* 1	/opt/jdk/jdk1.8.0_51/bin/java
+ 2	/opt/jdk/jdk1.8.0_66/bin/java
3	/opt/jdk/jdk-9/bin/java

Enter to keep the current selection[+], or type selection number: 3

To test if alternatives is pointing to the correct version, just type the following in your terminal window:

```
$ java -version
```

The following is the output:

```
java version "1.9.0-ea"
Java(TM) SE Runtime Environment (build 1.9.0-ea-b90)
Java HotSpot(TM) 64-Bit Server VM (build 1.9.0-ea-b90, mixed mode)
```

You'll also want to configure alternatives for your javac compiler by performing similar steps as before.

```
$ sudo alternatives --install /usr/bin/javac javac /opt/jdk/jdk-9/bin/javac 100
$ sudo alternatives --config javac
```

There are 3 programs which provide 'javac'.

Selection	Command
* 1	/opt/jdk/jdk1.8.0_51/bin/javac
+ 2	/opt/jdk/jdk1.8.0_66/bin/javac
3	/opt/jdk/jdk-9/bin/javac

To test if alternatives is pointing to the correct Java compiler version, just type the following:

```
$ javac -version
```

```
javac 1.9.0-ea
```

Ubuntu/Debian

When using you're Ubuntu or Debian Linux, you can obtain Oracle's Java JDK in two ways. The first way is to use the `apt-get` command and the second way is to download the JDK from the official site mentioned earlier.

With Ubuntu or Debian, you can use the `apt-get` command to download and install Oracle's official JDK 9:

```
$ sudo apt-get update
$ sudo apt-get install oracle-java9-installer
```

After, downloading Linux from Oracle's download site, you can manually decompress the `gz` file. For Ubuntu/Debian Linux distributions, pull up your terminal and type the following commands:

```
$ mv ~/Downloads/jdk-9-linux-x64.tar.gz /tmp
$ cd /tmp
$ tar zxvf jdk-9-linux-x64.tar.gz
$ sudo mv jdk-9 /usr/lib/jvm/
```

After installing JDK 9 on your Ubuntu/Debian system, you will likely encounter other versions of Java previously installed. Ubuntu/Debian-based operating systems have a way to switch between JDKs and JREs by using a tool called `update-alternatives`. Assuming you have installed the JDK 9 using `apt-get`, it usually puts the JDK in the `/usr/lib/jvm` directory. Now, you will want `alternatives` to register the new JDK 9 (`java` and `javac`).

Default Java

Type the following to register the new JDK in `alternatives`:

```
$ sudo update
$ sudo update-alternatives --install /usr/bin/java java /usr/lib/jvm/jdk-9-linux-x64/bin/java 100
```

Next, you'll want to set the newly registered `java` as the default version to use. Type in the following commands:

```
$ sudo update-alternatives --config java
```

There are 2 choices for the alternative `java` (providing `/usr/bin/java`).

Selection	Path	Priority	Status
* 0	/usr/lib/jvm/java-8-openjdk/jre/bin/java	1061	auto mode
1	/usr/lib/jvm/java-9-oracle/bin/java	100	manual mode

Press enter to keep the current choice[*], or type selection number: 1

```
$ sudo update-alternatives --display java
```

Selection	Path	Priority	Status
0	/usr/lib/jvm/java-8-openjdk/jre/bin/java	1061	auto mode
* 1	/usr/lib/jvm/java-9-oracle/bin/java	100	manual mode

Default Javac

You'll also want to switch the javac compiler too. To set the Java compiler, type the following commands to register the new javac:

```
$ sudo update-alternatives --install /usr/bin/javac javac /usr/lib/jvm/jdk-9-linux-x64/bin/javac 100
$ sudo update-alternatives --config javac
```

There are 2 choices for the alternative javac (providing /usr/bin/javac).

Selection	Path	Priority	Status
* 0	/usr/lib/jvm/java-8-openjdk/bin/javac	1061	auto mode
1	/usr/lib/jvm/java-9-oracle/bin/javac	100	manual mode

Press enter to keep the current choice[*], or type selection number: 1

```
$ sudo update-alternatives --config javac
```

There are 2 choices for the alternative javac (providing /usr/bin/javac).

Selection	Path	Priority	Status
0	/usr/lib/jvm/java-8-openjdk/bin/javac	1061	auto mode
* 1	/usr/lib/jvm/java-9-oracle/bin/javac	100	manual mode

Setting Environment Variables

If you aren't using the conveniences of alternatives to point to your default Java binaries, you can do it the old fashioned way, by setting up Java *environment variables*. There are other strategies to easily switch Java versions that I've not mentioned in this chapter, but understanding Java's environmental variables will allow you to install and develop Java applications on any operating system.

To get started you need to set and update a couple of important environment variables. How you set them and the values they should be set to vary depending on your choice of operating system. The two variables to set are the following:

- **JAVA_HOME**: Tells your operating system where the Java installation directory lives.
- **PATH**: Specifies where the Java executable directory resides. This environment variable lets the system search paths or directories containing executable files. Java executables reside in the bin directory under the **JAVA_HOME** home directory.

The following are the environment variables to set in the 32-bit Windows operating system:

```
set JAVA_HOME="C:\Program Files (x86)\Java\jdk-9"  
set PATH=%JAVA_HOME%\bin;%PATH%
```

Set the same variables in 64-bit Windows, but the values are slightly different:

```
set JAVA_HOME="C:\Program Files\Java\jdk-9"  
set PATH=%JAVA_HOME%\bin;%PATH%
```

Note Depending on the version of the JDK the `JAVA_HOME` path, the name will change according to the version that was installed. For example, if the next version is named JDK 9.0.1, the path would likely appear as `JAVA_HOME="C:\Program Files\Java\jdk-9.0.1"`. To see Java 9's JDK and JRE version naming scheme, go to the following links:

<http://openjdk.java.net/jeps/223> and
<https://blogs.oracle.com/java-platform-group/a-new-jdk-9-version-string-scheme>.

On Unix-like platforms such as MacOS X and Linux, your settings and how you make them depend on which *shell* you are using. The following examples show you how to set the needed variables for Bash, Bourne, and CSH shells, respectively.

```
# Mac OS X /bin/bash  
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk-9.jdk/Contents/Home  
export PATH=$PATH:$JAVA_HOME/bin  
  
# Linux bash, bourne shell environments /bin/bash  
export JAVA_HOME=/usr/java/jdk-9 export PATH=$PATH:$JAVA_HOME/bin  
  
#csh environments /bin/csh  
setenv JAVA_HOME /usr/java/jdk-9  
setenv PATH ${JAVA_HOME}/bin:${PATH}
```

These statements will set environment variables temporarily for the current terminal window session. To make `JAVA_HOME` and `PATH` more permanent, you will want to add them to your system upon logon such that they are always made available whenever you boot or log in. Depending on your operating system, you will need to be able to edit environment variable names and values.

Setup Windows Environment Variables

In the Windows environment you can use the keyboard shortcut Windows logo key+Pause/Break key, and then click the Advanced system settings (link on the left side) to display the Systems Properties window, as shown in Figure 1-12.

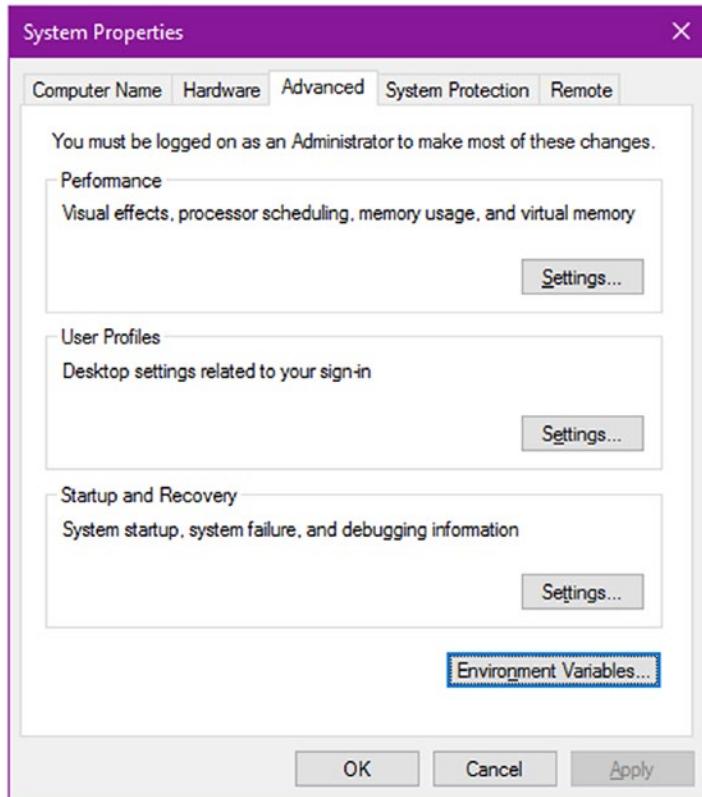


Figure 1-12. The System Properties window

Next, click the Environment Variables button. At this point you can add, edit, and delete environment variables. You will add or edit the JAVA_HOME environment variable by using the installed home directory as the value. The Environment Variables window on the Windows operating system is shown in Figure 1-13.

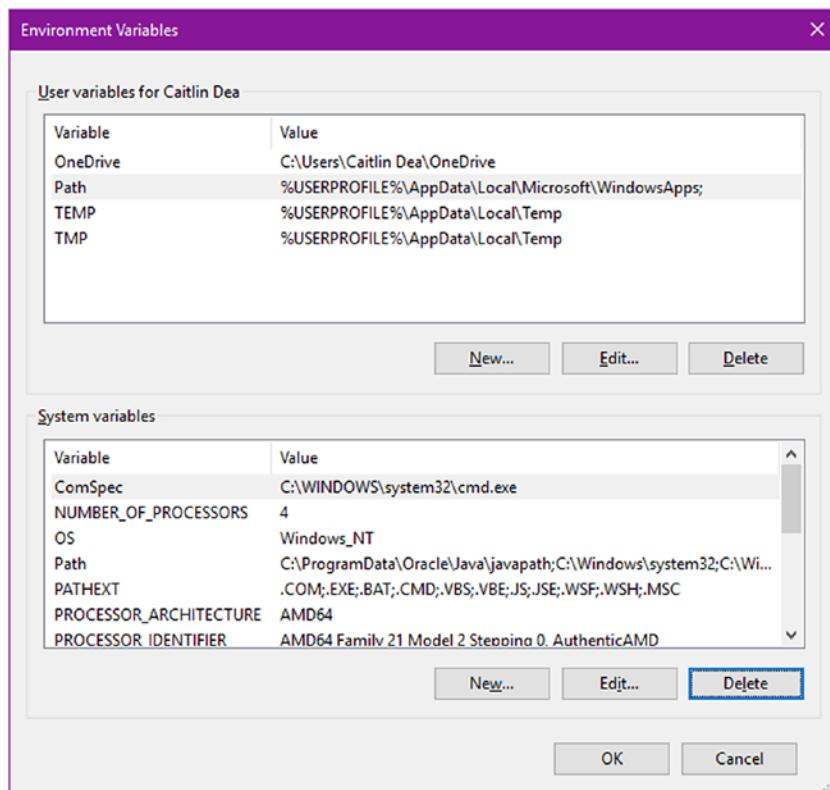


Figure 1-13. Windows Environment Variables window

To add the `JAVA_HOME` environment variable, click on the New button shown in Figure 1-13.

After you click the New button, the window in Figure 1-14 appears and allows you to enter the environment variable name and value. When you're done, click on the OK button.

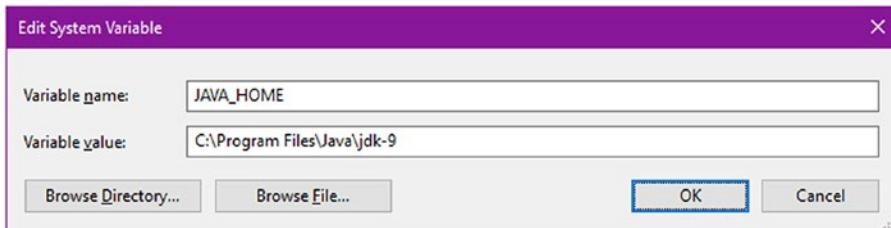


Figure 1-14. The Add/Edit window to enter the `JAVA_HOME` environment variable

After, you've clicked on OK to add `JAVA_HOME`, you will need update the Path environment variable, as shown in Figure 1-15. Select the Path environment variable in the System Variables area. Next, click on the Edit button.

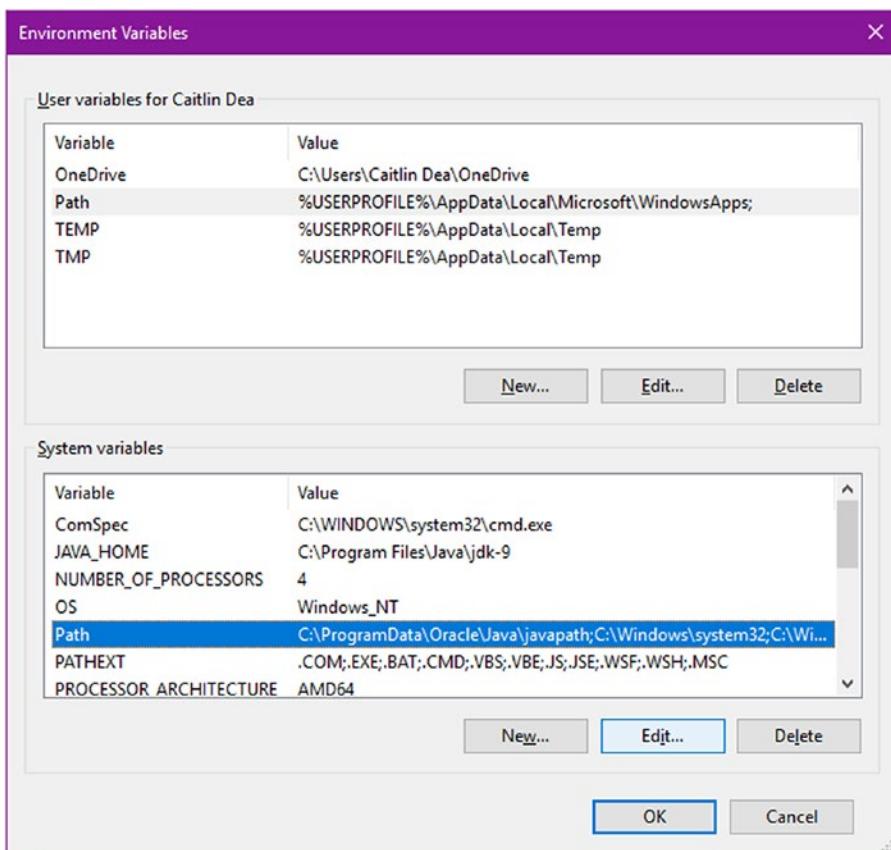


Figure 1-15. Editing the Path environment variable

In Windows 10, you need to edit the Path environment variable by adding a new entry, as shown in Figure 1-16. Here you will add an entry such as %JAVA_HOME%\bin. To add the entry, click on the New button. If the entry already exists, click on the Edit button. This allows the system to see all binary executables in the bin directory of the JAVA_HOME. In other words, when you are on the terminal or command-line prompt, the system will recognize files such as javac.exe for the Java compiler.

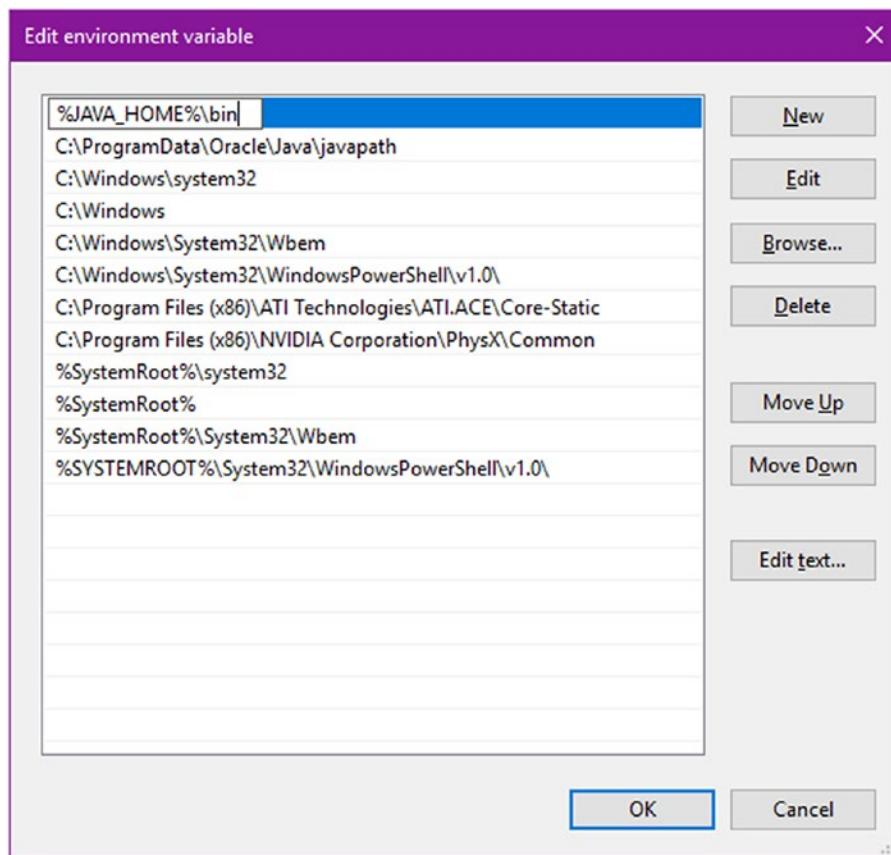


Figure 1-16. Adding %JAVA_HOME%\bin to the Path environment variable

In Figure 1-16, the %JAVA_HOME%\bin entry was at the bottom of the list, but I moved it to the top by clicking on the Move Up button.

Setting Up MacOS X/Linux Environment Variables

To make the environment variables permanent on MacOS X or Linux, you can use common text editors such as *nano* or *vim*. But before you go editing your user shell profile files, you will want to make a backup copy just in case you make a mistake. Because many Unix and Linux systems have different shells, I've detailed the typical files to copy and edit.

On a Mac, type the following:

```
$ cp ~/.bash_profile ~/.bash_profile.orig
```

On a Linux OS, you'll type the following to determine the kind of shell you have, then you'll know which file to copy and edit. Different shells use different files to export environment variables when creating a console session.

```
$ echo $0
-bash
```

If the output displays `-bash`, you'll want to back up and edit the `.bashrc` file from your home directory. If the output displays `-csh`, you'll want to back up and edit the `.cshrc` from your home directory. Next, you copy the `.bashrc` file as a backup before editing it.

```
$ cp ~/.bashrc ~/.bashrc.orig
```

If you're on a Linux OS using the C shell, type the following:

```
$ cp ~/.cshrc ~/.cshrc.orig
```

Now, you'll want to edit the file that you just backed up by using `nano` or `vi` text editor. To load a file, you type the following:

```
$ nano ~/.bash_profile
```

Or

```
$ vi ~/.bash_profile
```

Notice that I used the `~/.bash_profile` file located in the `home` directory on the MacOS X operating system. I trust you will replace the file with the particular file depending on your Linux OS and shell. After loading the file from the previous command you will want to perform edits, saves, and exits from those editors. Tables 1-1 and 1-2 are quick run-downs of the common commands for `nano` and `vim`, respectively.

Table 1-1. The Nano Editor's Keyboard Commands After Editing a File

Action	Command	Notes
Navigate	Use arrow keys up, down, right, and left	Arrow keys, Page Up/Down, and Home and End can also be used to navigate the cursor.
Save File	Ctrl/Control+O	Editor prompts the name of the file. Just press Enter to accept the same name.
Exit Editor	Ctrl/Control+X	Brings you back to the command-line prompt.

Table 1-2. The vi Editor's Keyboard Commands After Editing a File

Action	Command	Notes
Navigate	Use arrow keys up, down, right, and left	Arrow keys, Page Up/Down, and Home and End can also be used to navigate cursor.
Insert text	i	After a text file is loaded you can modify the file, but before you can, you have to press 'i' for insert. To get back to command mode, press the ESC key.
Add text	a	After a text file is loaded you can modify the file, but before you can you have to press 'a' for add. To get back to command mode, press the ESC key.
Command mode	ESC	Exits editor mode, bringing you to command mode.
Save file	:w <Enter key>	Assuming you are in command mode (press ESC) before typing :w (colon 'w'). The file is saved. To save and exit, type :wq.
Exit editor	:q	Brings you back to the command-line prompt.
Exit with no changes	:q!	Modifications are ignored and exits editor.
Save and Exit	:wq	File is saved and exits immediately.

- To set your JAVA_HOME for the MacOS X platform, you need to launch a terminal window to edit your home directory's .bash_profile file (for example, nano ~/.bash_profile) by adding the export commands at the bottom of the file:

```
# Mac OS X /bin/bash
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk-9.jdk/
Contents/Home
export PATH=$PATH:$JAVA_HOME/bin
```

- On Linux and other Unix operating systems that use Bash shell environments, launch a terminal window and edit either the ~/.bashrc or ~/.profile file to contain the export commands. Add the export commands.

```
# Linux bash, bourne shell environments /bin/bash
export JAVA_HOME=/usr/java/jdk-9
export PATH=$PATH:$JAVA_HOME/bin
```

- On Linux and other Unix operating systems that use C shell (csh) environments, launch a terminal window and edit either the ~/.cshrc or ~/.login file to contain the setenv commands. Add the export commands.

```
#csh environments
setenv JAVA_HOME /usr/java/jdk-9
setenv PATH ${JAVA_HOME}/bin:${PATH}
```

Once you've set up your path and `JAVA_HOME` environment variables, you will want to verify them by launching a terminal window and executing the following two commands from the command prompt:

```
$ java -version
$ javac -version
```

The output in each case should display a message indicating the Java 9 version of the language and runtime. Here is the display output of the version.

```
Java -version

java version "1.9.0-ea"
Java(TM) SE Runtime Environment (build 1.9.0-ea-b90)
Java HotSpot(TM) 64-Bit Server VM (build 1.9.0-ea-b90, mixed mode)

javac -version

javac 1.9.0
```

Installing Gradle

If you have not downloaded Gradle yet, visit <http://gradle.org/gradle-download>. After downloading the binary distribution, you'll want to install Gradle. To install Gradle, follow these steps:

1. Unzip the Gradle software into a directory.
2. Update the environment variable `GRADLE_HOME` to point to the directory located where the `gradle` folder was created after the unzipped file.
3. Update the environment variable `PATH` to include the `$GRADLE_HOME/bin`. In Windows, it would be `%GRADLE_HOME%/bin`. To make these variables permanent, follow the instructions mentioned earlier regarding `JAVA_HOME`.
4. To verify that Gradle was installed properly at the command line (the terminal), type the following command:

```
gradle -version
```

The version information is shown here:

```
Gradle 2.7
```

```
Build time: 2015-09-14 07:26:16 UTC
Build number: none
Revision: c41505168da69fb0650f4e31c9e01b50ffc97893

Groovy: 2.3.10
Ant: Apache Ant(TM) version 1.9.3 compiled on December 23 2013
JVM: 1.9.0-ea (Oracle Corporation 1.9.0-ea-b90)
OS: Mac OS X 10.11.1 x86_64
```

Now that you have the JDK installed, you can install the NetBeans IDE. Installing the NetBeans IDE is optional, so you may skip the next section.

Installing the NetBeans IDE

When developing JavaFX applications, you might want to use the NetBeans IDE. Be sure to download the correct NetBeans version containing JavaFX. If you haven't already downloaded the NetBeans IDE, go to following location:

<https://netbeans.org/downloads/index.html>

To install the NetBeans IDE, follow these steps:

1. Install NetBeans 8.2 or later. Launch the binary executable. On the Windows platform, you will receive a security warning to inform the user about installing software.
2. Begin the NetBeans IDE installation process by clicking the Next button. Figure 1-17 prompts the user to begin the installation process.

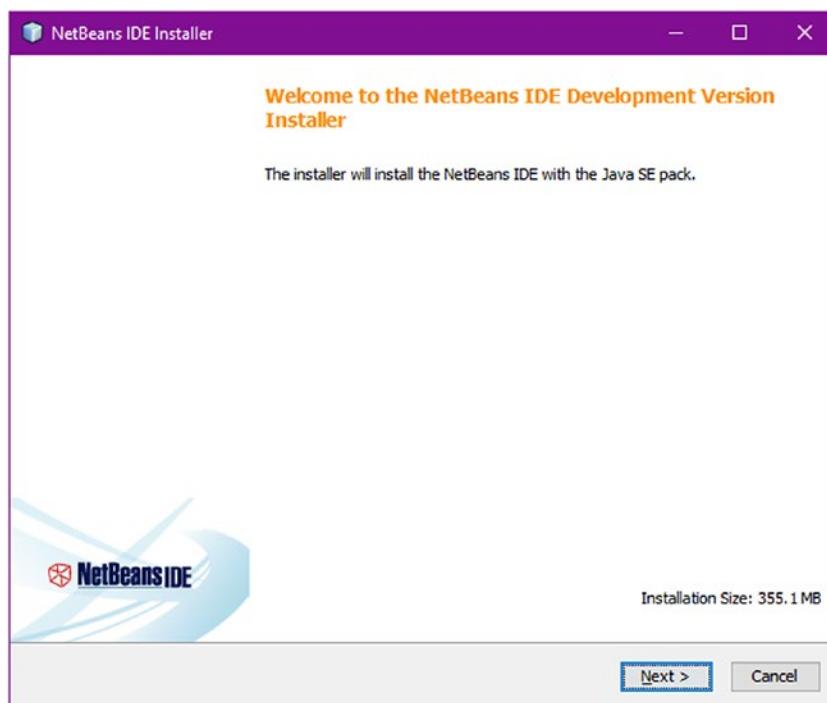


Figure 1-17. NetBeans IDE installer

3. Read and accept the NetBeans license agreement. After reading and agreeing to the terms, click the “I accept the terms in the license agreement” check box and then click Next to proceed as shown in Figure 1-18.

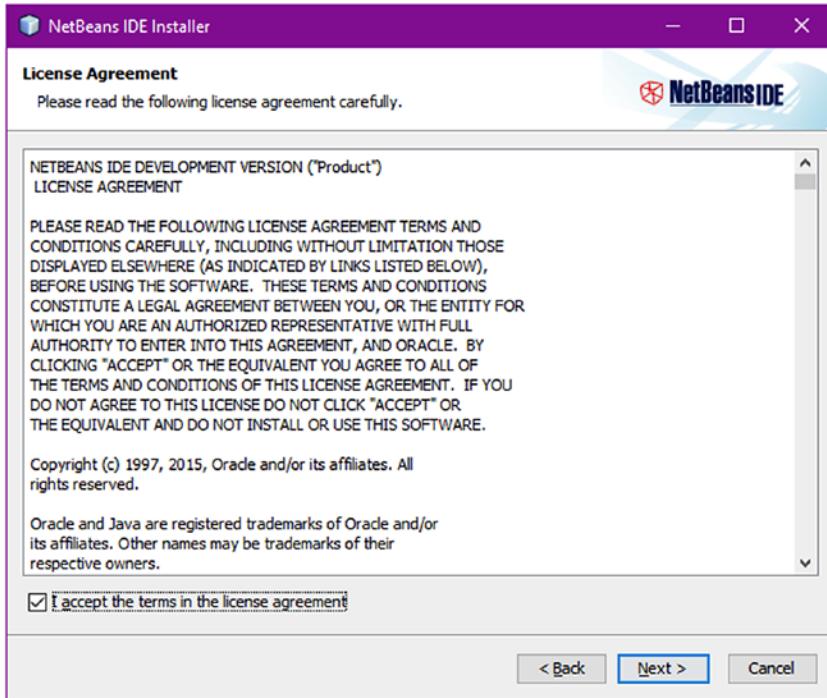


Figure 1-18. NetBeans license agreement

4. Next, you can choose the destination directory to install NetBeans and select the Java 9 SE Development Kit previously installed. As usual, keep the defaults for the destination to install the NetBeans IDE and the JDK 9, as shown in Figure 1-19.

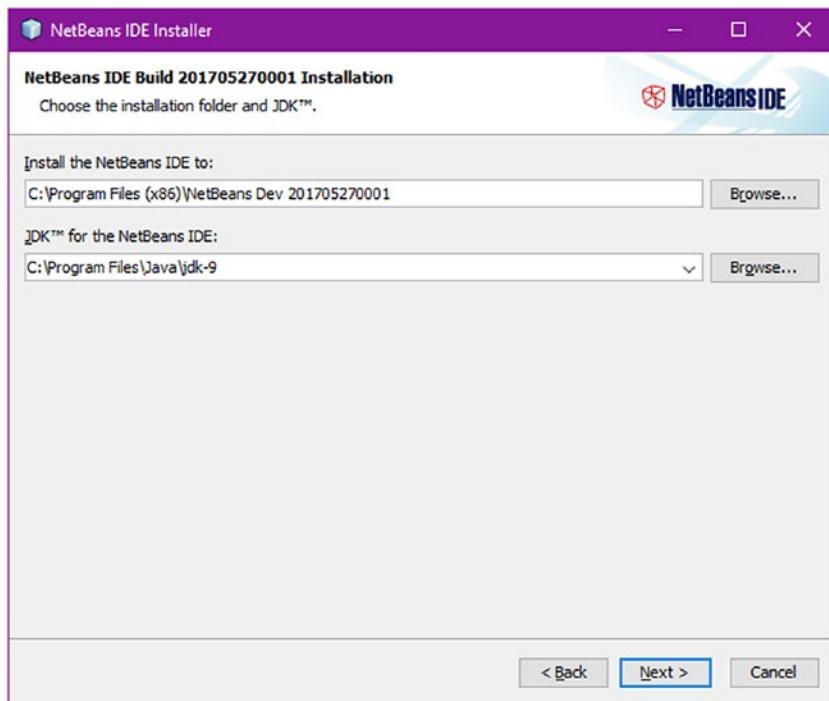


Figure 1-19. NetBeans IDE installation

5. On the Summary screen, the Check for Updates option allows the installer to retrieve any updates or patches to the current NetBeans version and any other plug-in dependencies. Accept the defaults and click the Install button, as shown in Figure 1-20.

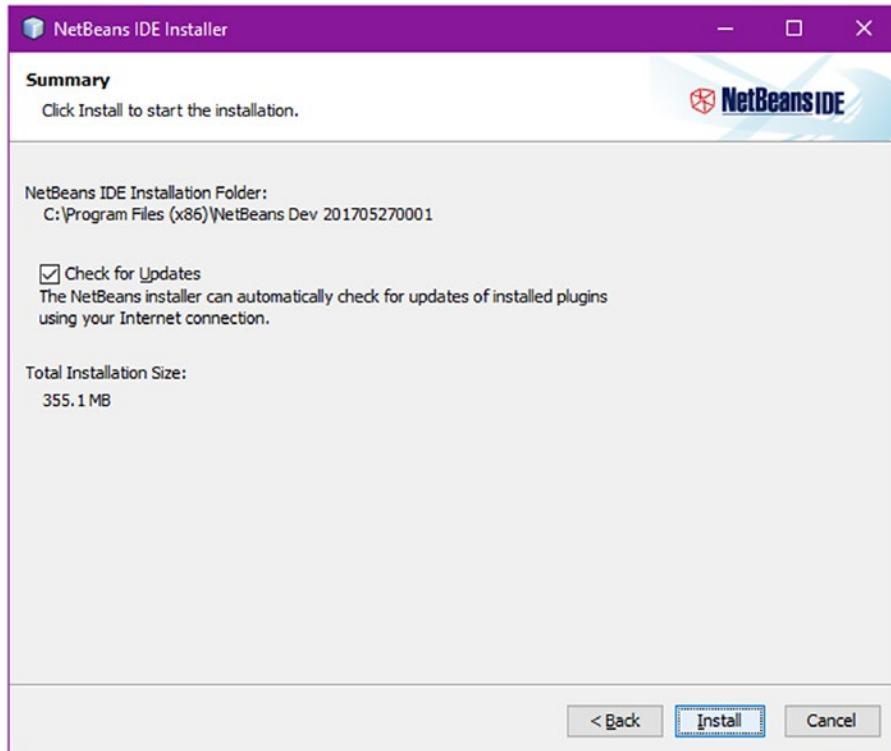


Figure 1-20. Check for updates

Figure 1-21 shows the installation progress bar.

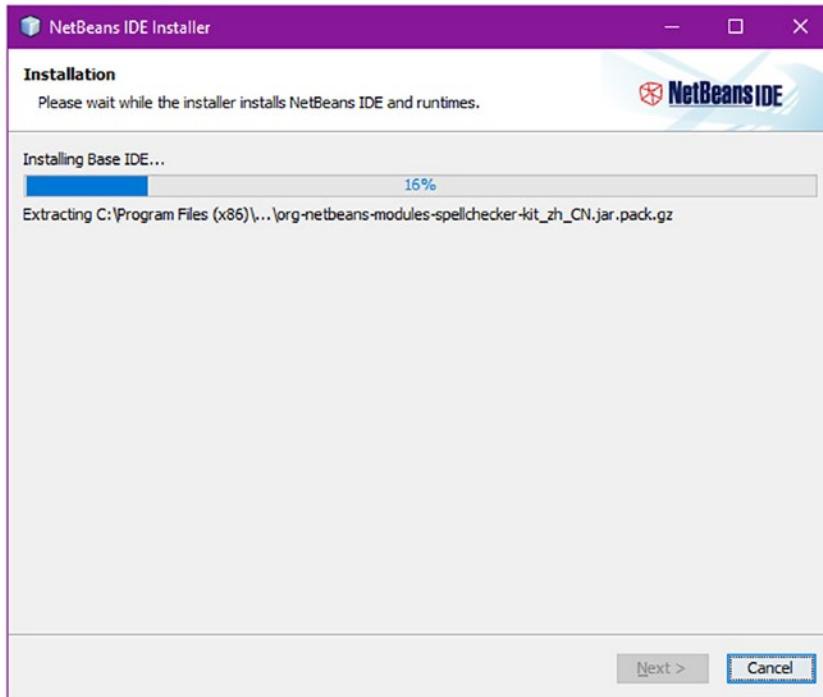


Figure 1-21. Installation progress

6. To complete the setup, in Figure 1-22 you will be prompted for an optional check box to Contribute to the NetBeans team to provide anonymous data usage to help diagnose issues and enhance the product. Once you have decided whether to contribute, click the Finish button to complete the installation.

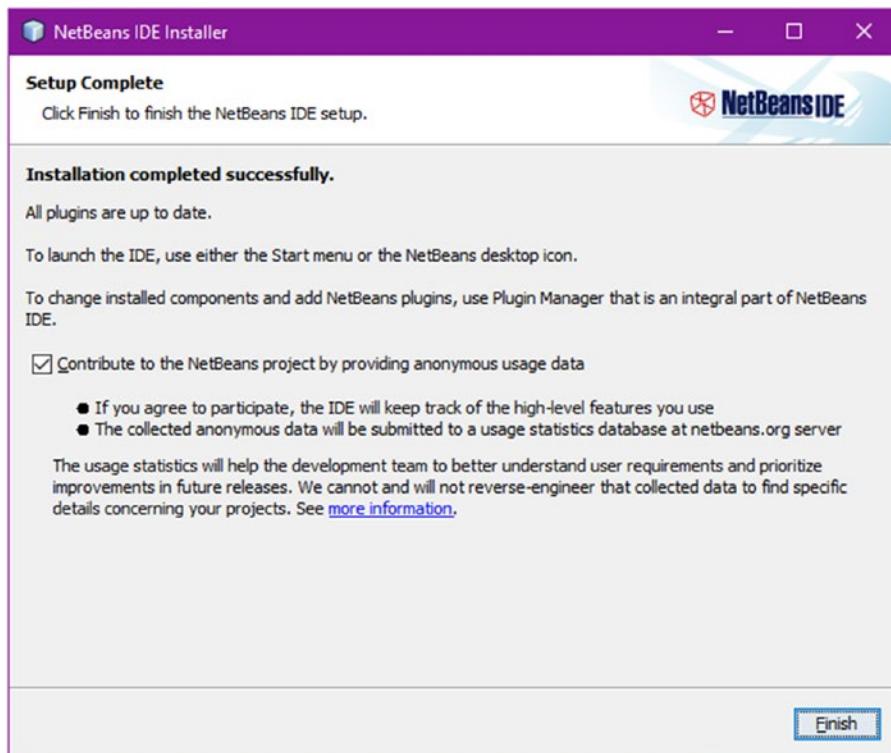


Figure 1-22. Setup complete

7. Launch the NetBeans IDE to see the opening page, as shown in Figure 1-23.

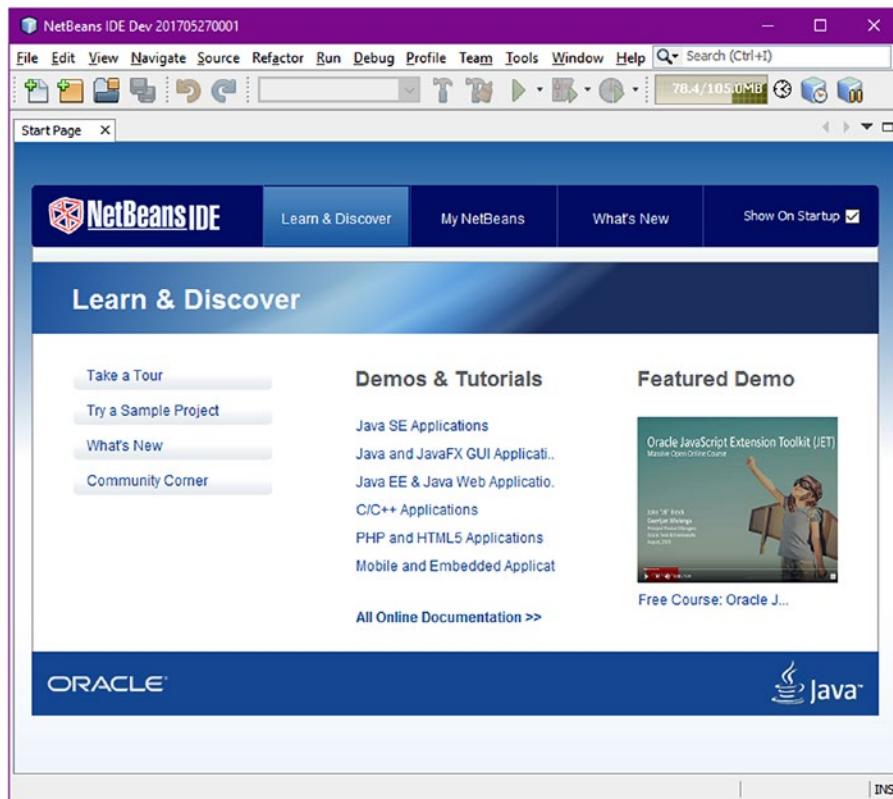


Figure 1-23. NetBeans start page

Now you are ready to go forward and create awesome JavaFX applications!

Creating a JavaFX HelloWorld Application

When creating JavaFX applications, you'll be learning three ways to develop, compile, and run applications.

- NetBeans IDE
- Editor and terminal (the command-line prompt)
- Editor and the Gradle Build tool

Using the NetBeans IDE

To quickly get started with creating, coding, compiling, and running a simple JavaFX HelloWorld application using the NetBeans IDE, follow the steps outlined in this section. You'll want to go deeper in the next section to get an understanding of what the IDE is doing behind the scenes on your behalf. For now, though, let's just get some code written and running.

1. Launch NetBeans IDE.
2. On the File menu, select File ► New Project.
3. Under Choose Project and Categories, select the JavaFX folder.
4. Under Projects, select Java FX Application, and click Next.
5. Specify `HelloWorld` for your project name.
6. Change or accept the defaults for the Project Location and Project Folder fields.

Figure 1-24 shows the New JavaFX Application wizard used to create a simple HelloWorld application.

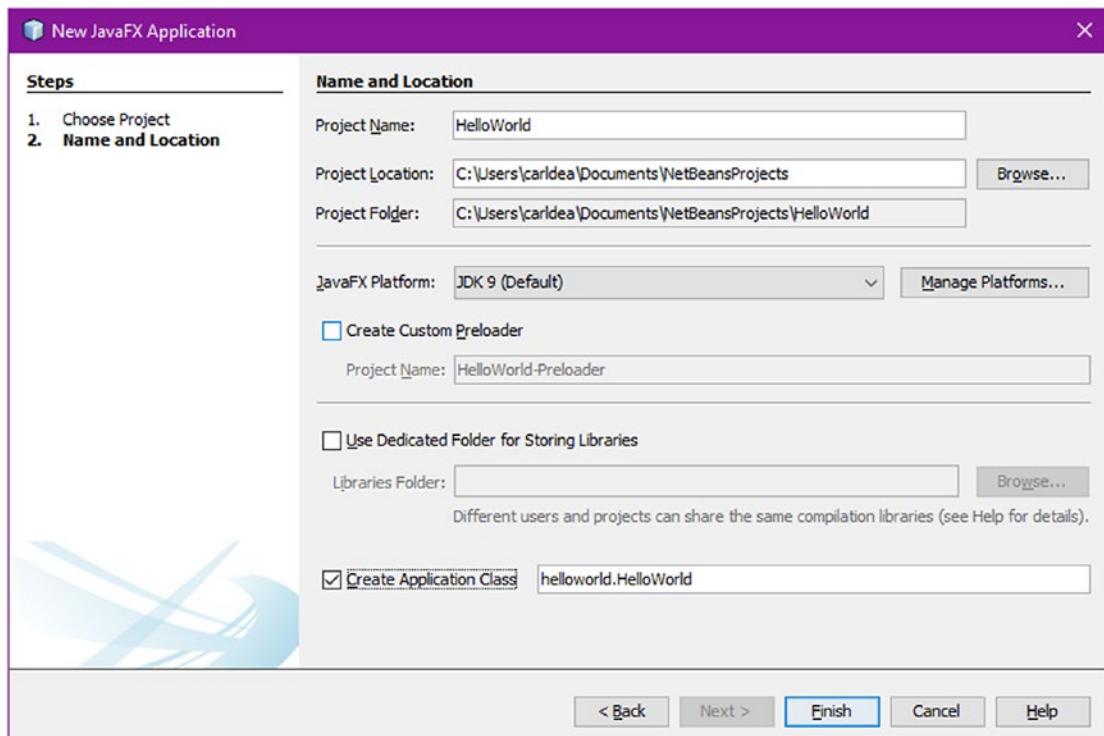


Figure 1-24. New JavaFX Application wizard

7. On this screen (Figure 1-24), click the Manage Platforms button to make sure the Platform Name field contains JDK 1.9 or greater as the default. Click Finish to auto-create the JavaFX HelloWorld application.
8. In a few seconds, NetBeans will generate all the necessary files for the HelloWorld project. After NetBeans finishes, the project will show up on the left Projects tab. Figure 1-25 shows a newly created HelloWorld project.

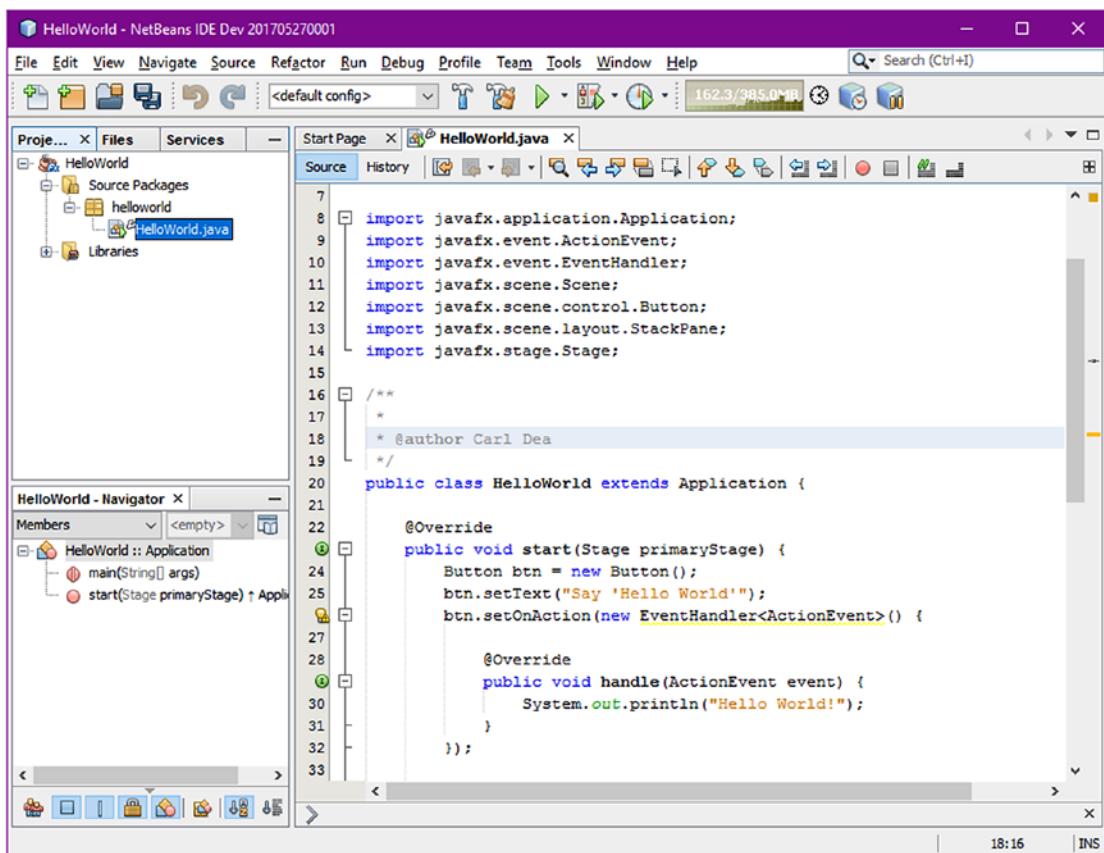


Figure 1-25. A newly created HelloWorld project

9. In the NetBeans IDE on the Projects tab (left), select the newly created project. Open the Project Properties dialog box from the File menu option Project Properties. Here you will verify that the Source/Binary format setting is using JDK 9, as shown in Figure 1-26. Click Sources under Categories.

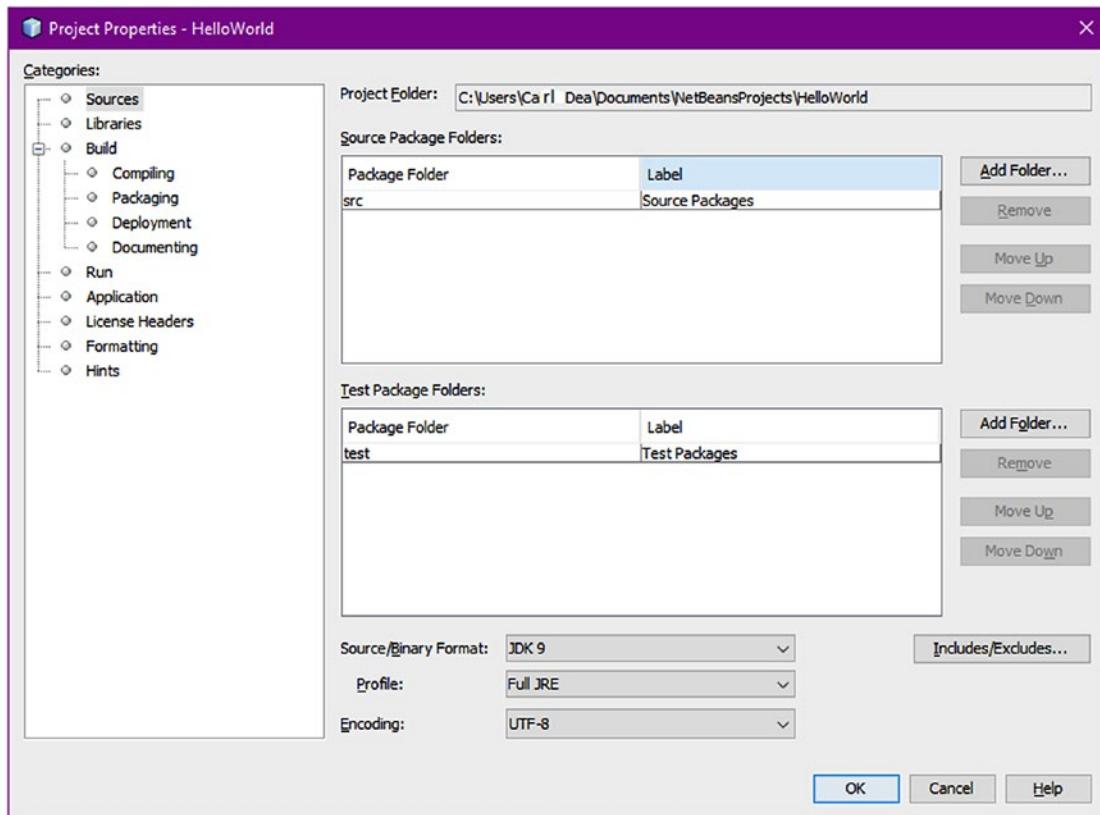


Figure 1-26. Project properties

10. To run and test your JavaFX HelloWorld application, ensure the project folder is selected (the left Projects tab), then click on the green Run button or press the F6 key to execute the HelloWorld project. Figure 1-27 shows the Run Project option used to launch the JavaFX application.

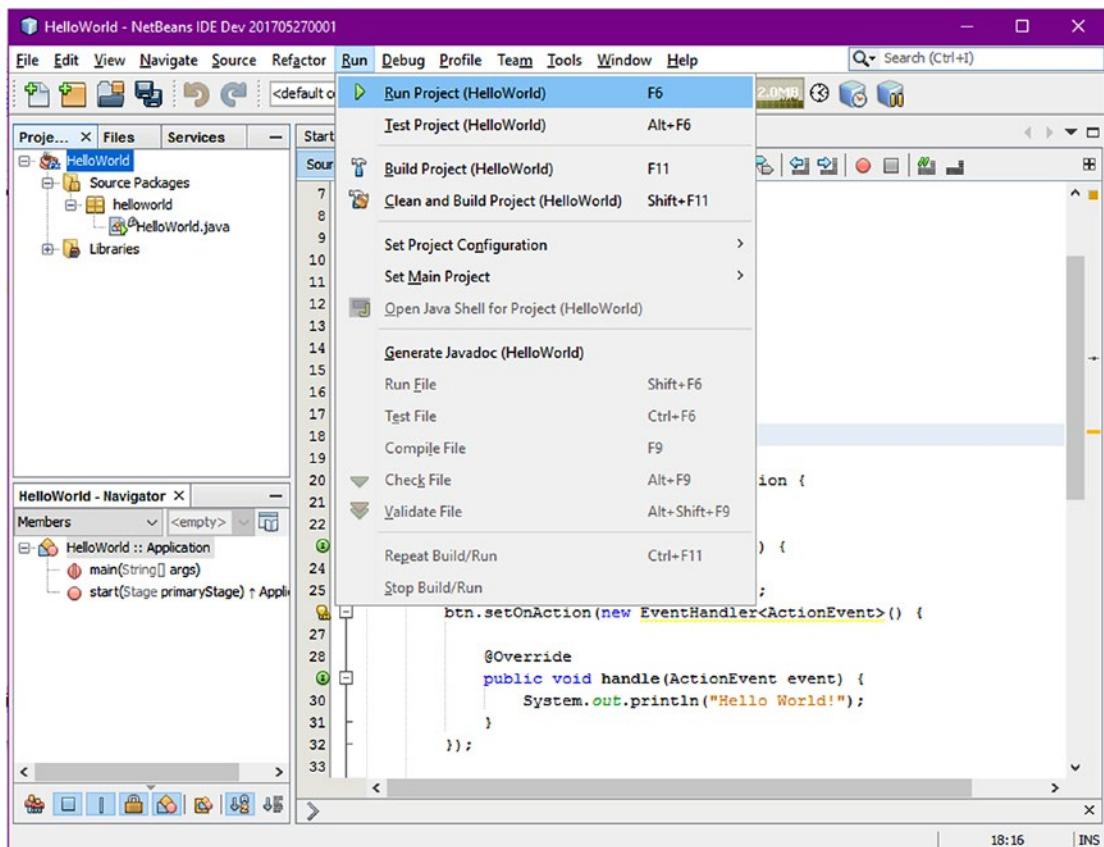


Figure 1-27. Running the HelloWorld project

11. After you choose the Run option, the output should look like Figure 1-28.

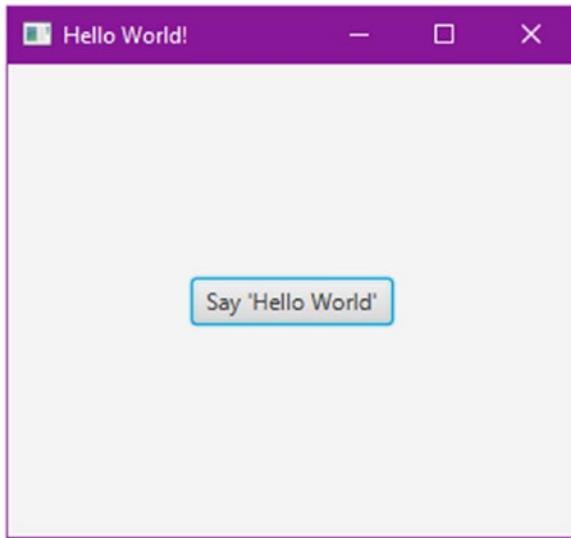


Figure 1-28. JavaFX HelloWorld launched from the NetBeans IDE

You shouldn't encounter any difficulty when following Steps 1 through 8. However, steps 9 ensures that your project's source/binary format is using JDK 9. Because of the new additions to the Java language in Java 9, such as modules, some of the source code in this book will rely on the new syntax and therefore will not be backward-compatible with prior versions of Java 9. Typically, developers have prior versions of the Java development kit such as JDK 8. NetBeans allows you to switch between different JDKs when projects require older versions. An important thing to note is that in early releases of JavaFX 2.0, the software was packaged separately from the Java SDK, which caused some software versioning confusion. Thank goodness it is now just one download (JDK 9)! If you are still using Java 7 SDK, the latest download will also contain JavaFX in one bundle since JavaFX 2.1.

Note Did you know that Java 7 is at the end of its life? This means that Oracle will no longer post any updates or releases for the public. Unless you still have a commercial license and a support contract with Oracle, it is advised you migrate applications to Java 8 or later. Having no updates to Java 7's runtime may be a security concern. So, you've been warned. The official Java 7 end of life and FAQ site is here:
https://www.java.com/en/download/faq/java_7.xml.

Using an Editor and the Terminal (the Command-Line Prompt)

The second way of developing JavaFX 9 applications is to use a common text editor and the command-line prompt (terminal). By learning how to compile and execute applications on the command-line prompt, you will learn about the *classpath* and where the executable files reside. Being exposed to this will sharpen your skills when you are in environments where fancy IDEs and/or editors aren't easily available.

Working from the command line, you will basically use a text editor such as *nano*, *vi*, *Emacs*, or *Notepad* to code your JavaFX HelloWorld source. The example shown in Listing 1-1 is of a HelloWorld Java application.

Listing 1-1. A JavaFX HelloWorld Application from HelloWorld.java

```
package org.acme.helloworld;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;

/**
 * A JavaFX Hello World Application
 */
public class HelloWorld extends Application {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        stage.setTitle("Hello World");
        Group root = new Group();
        Scene scene = new Scene(root, 300, 250);
        Button btn = new Button();
        btn.setLayoutX(100);
        btn.setLayoutY(80);
        btn.setText("Hello World");
        btn.setOnAction(new EventHandler<ActionEvent>() {

            public void handle(ActionEvent event) {
                System.out.println("Hello World");
            }
        });
        root.getChildren().add(btn);
        stage.setScene(scene);
        stage.show();
    }
}
```

Once the Java file is created you will use the command-line prompt or terminal window to compile and run your JavaFX application. The following are steps to create a JavaFX HelloWorld application to be compiled and run on the command-line prompt (terminal).

1. In your user home directory, make a `projects` directory and underneath it you will create a `helloworld` directory.

On a MacOS or Linux operating system, use the following command to make a `projects` directory:

```
$ mkdir -p ~/projects/helloworld/src  
$ mkdir ~/projects/helloworld/classes  
$ cd ~/projects/helloworld/  
$ nano src/HelloWorld.java
```

On a Windows operating system, use the following command to change directories:

```
C:\Users\myusername>mkdir projects  
C:\Users\myusername>mkdir projects\helloworld  
C:\Users\myusername>mkdir projects\helloworld\src  
C:\Users\myusername>mkdir projects\helloworld\classes  
C:\Users\myusername>cd projects\helloworld  
C:\Users\myusername\projects\helloworld>notepad src\HelloWorld.java
```

2. Copy and paste the code from Listing 1-1 into a text editor and save the file as `HelloWorld.java`. The file should reside inside the `src` directory.

Note When entering the code for the `HelloWorld.java` file from Listing 1-1, be sure to include the package `org.acme.helloworld;` statement at the top. By doing so, the package namespace directories are created when the code is compiled. For example, the compiled `HelloWorld.class` file will reside in the `classes/org/acme/helloworld` directory.

3. Compile the source code file called `HelloWorld.java` using the Java compiler `javac` with the `-d` switch and the `classes` directory. The `classes` directory is where the compiled code goes. Also, you need to specify the source code files to compile, which is `src/*.java`. The commands you need to enter depend on your operating system.

On a MacOS X or Linux operating system, type the following on the command-line prompt:

```
$ javac -d classes src/*.java
```

On a Windows command line, type the following:

```
C:\Users\myusername\projects\helloworld> javac -d classes src\*.java
```

Notice the `-d classes` before the filename. The `-d` option (destination directory) lets the Java compiler know where to put the compiled class files based on their package namespace. In this scenario, the `HelloWorld` package statement (namespace) is `org.acme.helloworld`, which will create subdirectories under the `classes` directory, assuming you are in the `~/projects/helloworld` project directory.

When you're finished compiling, your directory structure should resemble Figure 1-29.

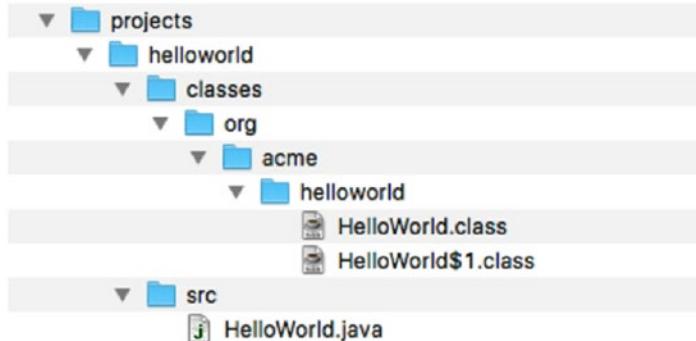


Figure 1-29. The compiled `HelloWorld` example creates directories based on the package namespace under the `classes` directory. You will notice the subdirectories are created as `org/acme/helloworld` under the `classes` directory you created earlier.

- Run and test your JavaFX `HelloWorld` application. Assuming you are in the `~/projects/helloworld` directory, type the following command to run your JavaFX `HelloWorld` application from the command-line prompt:

```
$ java -cp classes org.acme.helloworld.HelloWorld
```

On Windows OS:

```
C:\Users\myusername\projects\helloworld> java -cp classes org.acme.helloworld.HelloWorld
```

By using the `-cp`, `classes` will let Java know where the classpath directory is located. Figure 1-30 shows the output of a simple JavaFX `HelloWorld` application launched from the command-line prompt.

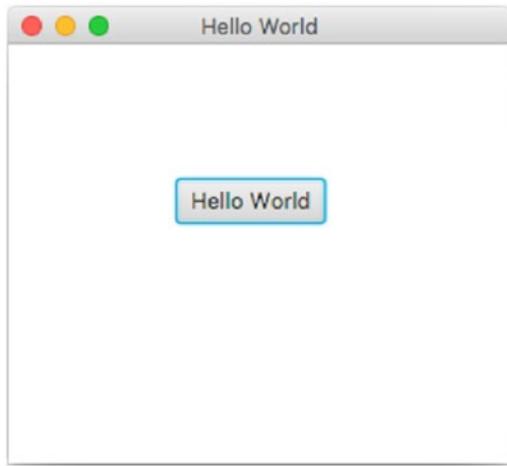


Figure 1-30. JavaFX HelloWorld launched from the MacOS X command-line prompt

Using Gradle on the Command-Line Prompt

A third way of developing JavaFX applications is using the Gradle *Build* tool. Here you will be also using a common text editor like before, but instead of manual steps in creating and compiling code, you will be relying on the Gradle Build tool. If you have not set up Gradle, refer to the early part of this chapter in the section entitled “Installing Gradle.” Follow these steps to create a Gradle-based project.

1. Create another project and directory named `helloworld_gradle`. Assuming you are in the `~/projects` directory, you will create a Gradle-based project.

On a Unix-like OS, enter the following:

```
$ mkdir -p projects\helloworld_gradle
```

On Windows, enter the following:

```
C:\Users\myusername>mkdir projects\helloworld_gradle
```

2. Change your directory to the `helloworld_gradle`.

On a Unix-like OS, enter the following:

```
$ cd helloworld_gradle
```

On Windows, enter the following:

```
C:\Users\myusername>cd projects\helloworld_gradle
```

3. Build a Java skeleton project having the conventional directory structure (Maven and Gradle convention). You will notice directories created that will look like `src/main/java` and `src/test/java`. Since this creates the project files and directories initially, it should only be done once. The following command creates a Java library project:

```
$ gradle init --type java-library
```

The following is the output when executing the Gradle `init` task:

```
:wrapper
:init
```

```
BUILD SUCCESSFUL
```

```
Total time: 1.897 secs
```

4. Since you are creating a JavaFX application and not a Java library, you will remove the `Library.java` and `LibraryTest.java` files under the `src/main/java` and `src/test/java` directories, respectively. These files are just stubbed out to get you started if you decide to create a Java library. In this scenario, you are creating a JavaFX application.
5. Next, you'll create a directory called `org/acme/helloworld` under the `src/main/java` directory and copy the `HelloWorld.java` file from Listing 1-1 into the `src/main/java/org/acme/helloworld` directory. This will look like Figure 1-31.

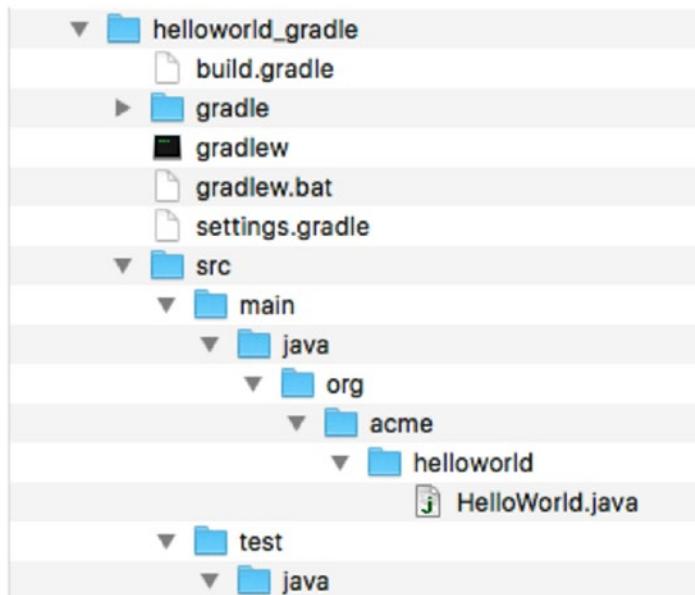


Figure 1-31. Gradle Java application project structure. Notice the standard convention of source code directory structure laid out as `src/main/java` and `src/test/java`. Build tools such as Maven and Gradle follow this convention.

6. Since you are creating a JavaFX application, you will need to edit the `build.gradle` file by adding a plug-in that will build the Java project as a launchable application. You'll also add one more entry to let Gradle know which class the main application should be run as. Using your text editor, add the following entries to the `build.gradle` file:

```
apply plugin: 'application'

mainClassName = "org.acme.helloworld.HelloWorld"
```

- After you've saved and exited the file, you'll want to build the project using the `gradle assemble` command. This will create an executable jar file that can be run. Type the following to compile and build a HelloWorld jar executable.

```
$ gradle assemble
```

The following is the output after executing the Gradle assemble task.

```
:compileJava
Download https://jcenter.bintray.com/org/slf4j/slf4j-api/1.7.12/
slf4j-api-1.7.12.pom
Download https://jcenter.bintray.com/org/slf4j/slf4j-parent/1.7.12/
slf4j-parent-1.7.12.pom
Download https://jcenter.bintray.com/org/slf4j/slf4j-api/1.7.12/
slf4j-api-1.7.12.jar
:processResources UP-TO-DATE
:classes
:jar
:assemble

BUILD SUCCESSFUL

Total time: 4.458 secs
```

- Now run the HelloWorld application by using the following command (run task).

```
$ gradle run
```

- The Gradle run task was added when you added the plug-in entry `apply plugin: 'application'`. Gradle plug-ins provide convenient ways to build different kinds of Java projects and artifacts. You should see the Java HelloWorld application launch just like in Figure 1-30.

There are many more powerful things you can do with Gradle that are beyond the scope of this book; however, I will list some common commands you might be interested in using.

- `gradle tasks`: Lists all the available tasks
- `gradle clean`: Removes directories and files from the build directory
- `gradle eclipseProject`: Creates project files for the Eclipse IDE. This task will be available when you add the following statement to the `build.gradle` file:

```
apply plugin: 'eclipse'
```

- `gradle intellijProject`: Creates project files for the IntelliJ IDE. This task will be available when you add the following statement to the `build.gradle` file:

```
apply plugin: 'idea'
```

- `gradle -help`: Shows the available options and descriptions of commands

Walking Through the HelloWorld Source Code

You'll notice in the source code that JavaFX-based applications extend (inherit) from the `javafx.application.Application` class. The `Application` class provides application lifecycle functions such as initializing, launching, starting, and stopping during runtime. This provides a mechanism for Java applications to launch JavaFX GUI components separate from the main thread. The code in Listing 1-2 is a skeleton of the JavaFX HelloWorld application, having a `main()` method and an overridden `start()` method.

Listing 1-2. A Skeleton Version of the `HelloWorld.java` File

```
public class HelloWorld extends Application {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // On the main thread
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // On the JavaFX application thread

        // JavaFX code goes here...
    }
}
```

Here, in the `main()` method's entry point, you launch the JavaFX application by simply passing in the command-line arguments to the `Application.launch()` method. To access any arguments passed into the `launch()` method, you can invoke the `getParameters()` method of the `Application` class. See the Javadoc documentation for details on various ways to access named and raw arguments. The following link is the Javadoc documentation for the `getParameters()` method:

<https://docs.oracle.com/javase/8/javafx/api/javafx/application/Application.html#getParameters-->

After the `Application.launch()` method has executed, the application will enter a ready state, and the framework internals will invoke the `start()` method to begin. At this point, the program execution occurs on the JavaFX application thread and not on the main thread. When the `start()` method is invoked, a JavaFX `javafx.stage.Stage` object is available for the developer to use and manipulate. Following is the overridden `Application.start()` method:

```
@Override
public void start(Stage stage) {...}
```

When the program begins in the `start()` method, a separate thread of execution occurs, called the *JavaFX application thread*. Keep in mind that running on the JavaFX application thread is synonymous in concept with running on Java Swing's event dispatch thread. Just be advised the difference between the Java main thread and the UI thread when it relates to UI applications. Later in this book, you will learn how to

create background processes to avoid blocking the JavaFX application thread. When you know how to build applications to avoid blocking the GUI, the user will notice that your application is much more responsive (snappy) under heavy usage. Mastering responsiveness in GUI applications is an important concept in enhancing usability and the overall user experience. To see the Java Application API, visit the Javadoc here:

<https://docs.oracle.com/javase/8/javafx/api/javafx/application/Application.html>

<https://docs.oracle.com/javase/9/>

JavaFX Scene Graph

You'll notice that some classes and objects are oddly named, such as Stage and Scene. There isn't a coincidence—the designers of the API have modeled things after the idea of a theater or a play in which actors perform in front of an audience. In this analogy, a play consists of one-to-many scenes in which actors perform. And, of course, all scenes are performed on a stage one at a time. In JavaFX, the Stage is equivalent to an application window similar to Java Swing API JFrame or JDialog on the desktop. Depending on the device, such as a Raspberry Pi (Raspbian), there may be only one stage. You can think of a Scene object as a content pane, similar to Java Swing's JPanel, capable of holding zero-to-many Node objects (children).

Proceeding with the example, in the start() method, you see that for a JavaFX desktop window (stage) you can set the title bar using the setTitle() method. Next, you create a root node (Group), which is added to the Scene object as the top-level surface for the application window. The following code snippet shows you how to set the title and create the scene:

```
stage.setTitle("Hello World");
Group root = new Group();
Scene scene = new Scene(root, 300, 250);
```

JavaFX Node

A JavaFX *node* is a fundamental base class for all scene graph nodes to be rendered. The following graphics capabilities can be applied to nodes: *scaling, transforms, translations, and effects*.

Some of the most commonly used nodes are UI controls and Shape objects. Similar to a tree data structure, a scene graph will contain children nodes by using a container class such as the Group or Pane class. You'll learn more about the Group class later when you look at the ObservableList class, but for now you can think of them as Java List or Collection classes that are capable of holding child node objects. In the following code, a button (Button) node is created to be positioned on the scene and set with an event handler (EventHandler<ActionEvent>), which responds when the user presses the button. The handler code will output the text "Hello World" on the console. For now the code shown here uses an anonymous inner class; however you will learn later in Chapter 3 to use lambda expressions introduced in Java 8.

```
Button btn = new Button();
btn.setLayoutX(100);
btn.setLayoutY(80);
btn.setText("Hello World");
btn.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        System.out.println("Hello World");
    }
});
root.getChildren().add(btn);
```

Once the child nodes have been added to the root Group via the `getChildren().add()` method, you set the stage's scene and call the `show()` method on the Stage object to display a JavaFX application window. By default, the window will allow users to minimize, maximize, and close (exit) the application. The following code sets the scene and displays the JavaFX application window (the Stage):

```
stage.setScene(scene);
stage.show();
```

Packaging a JavaFX Application

At some point you will want to distribute or deploy your JavaFX application. To handle the numerous application packaging and deployment strategies, Oracle's Java team has created a Java Packager tool to assist developers in building, packaging, and deploying their applications. To learn more about how to use Java Packager tool, see Oracle's "Deploying JavaFX Applications" at the following URL:

<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/javapackager.html>

To give you a taste of the Java Packager tool's capabilities, I will show you how to package the HelloWorld classes into a single executable file for the many supported platforms. This will allow users to double-click to launch or install the application on the native operating system. The following commands will automatically compile and generate executables, launchers, and installers located in `~/projects/helloworld/dist/bundles` directory. Type the following to create deployable Java applications.

```
$ cd ~/projects/helloworld
$ javapackager -makeall -appclass org.acme.helloworld.HelloWorld -name hellothere
```

Table 1-3 describes the common options and switches that are used in building distributable JavaFX executable application bundles.

Table 1-3. Common JavaPackager Options to Build an Executable jar

Option/Switch	Example	Description
<code>-createjar</code>	<code>--</code>	Creates a JavaFX jar executable application.
<code>-appclass</code>	<code>org.acme.helloworld.HelloWorld</code>	Specifies the fully qualified name of the class containing the <code>main()</code> method.
<code>-srcdir</code>	<code>.</code>	The top-level location of the parent directory holding the compiled classes (current directory).
<code>-outdir</code>	<code>out</code>	The destination where the packaged jar file will be created.
<code>-outfile</code>	<code>helloworld.jar</code>	Specifies the name of the executable jar file.
<code>-v</code>	<code>--</code>	Allows verbose displays logging information when executing <code>javapackager</code> .
<code>-native</code>	<code>dmg</code>	JavaPackager can create installer, image, exe, msi, dmg, rpm, and deb file extensions as deployable executables on the native OS platform.
<code>-makeall</code>		Based on the OS the packager tool will build installers and executables. For example, for the MacOS X platform, the <code>dist/bundle</code> directory will contain executables with file extensions of the following: jnlp, dmg, app, pkg, and jar.

To run the jar executable on the command line, you simply type the following and press Enter:

```
$ java -jar dist/bundles/dist.jar
```

To launch HelloWorld as a standalone application on the MacOS X operating system, just double-click on the hellothere-1.0.dmg or hellothere-1.0.pkg application file in the bundles directory, as shown in Figure 1-32.

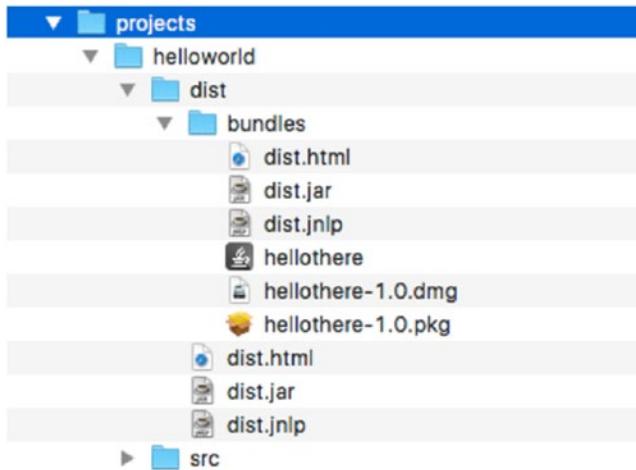


Figure 1-32. Java Packager creates executables and installers of the HelloWorld application named hellothere. Other application bundles exist, such as hellothere-1.0.dmg, to install the application correctly on the MacOS X desktop.

There are many ways to package and deploy JavaFX applications. To learn more, see Oracle's "Java Packager Guide" at the following URL:

<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/javapackager.html>

Also see the following, related articles:

- “Lesson: Deploying Self-Contained Applications” <http://docs.oracle.com/javase/tutorial/deployment/selfContainedApps/index.html>
- “Self-Contained Application Packaging” <https://docs.oracle.com/javase/8/docs/technotes/guides/deploy/self-contained-packaging.html>
- “Java Web Start Application” <https://docs.oracle.com/javase/tutorial/deployment/webstart/index.html>

Downloading the Book’s Source Code

To get the source code for the book’s examples, you can go to two places:

- Apress book’s site at <http://www.apress.com/us/services/source-code>
- GitHub at <https://github.com/carldea/jfx9be>

Summary

So far you have managed to download and install Java 9 JDK, Gradle, and NetBeans IDE. After successfully installing the prerequisite software, you created a JavaFX HelloWorld GUI application through the NetBeans IDE. Then you were able to use a text editor to enter your source code and eventually compile and run the binary class via a command-line prompt (the terminal window). Another method you learned was the use of the Gradle Build tool to compile and launch the JavaFX application using tasks.

After learning three methods for compiling, building, and running a JavaFX application, you went through a quick code walkthrough of the `HelloWorld.java` source file. You also learned to package a JavaFX application as a standalone `JAR`, `dmg`, `pkg`, and `JNLP` executable using the JavaFX Packager tool. Lastly, you learned that you have two places to obtain the source code for the book.

Next, in Chapter 2, you learn what's new in JDK 9 regarding project Jigsaw. If you understand JDK 9's modularity and are impatient, you can go to Chapter 3 to learn the fundamentals of JavaFX 9, such as drawing and coloring shapes as well as drawing text and changing text fonts.

CHAPTER 2



JavaFX and Jigsaw

In the early days of software architectures, many designers and practitioners often stressed the idea of *modular code*. Clearly, it makes sense to break up software services or APIs into modules; however, depending on who you talk to, the word *module* has different meanings to different people. For example, developers using Apache Maven build tools will often view jar artifacts as modules, while some would argue that packages within artifacts are also modules. So what does the word “module” mean in the Java world?

In this chapter, you’ll get to explore what modularity means for the new Java 9 platform. You’ll also learn the Java platform module system (JPMS) in the context of JavaFX applications. This chapter is a gentle introduction to Java 9’s module system with common use case scenarios such as using a module as a library dependency.

Having said this, I will not be going into great detail covering *every* modularity use case (believe me, there are quite a few of them). Even as of this writing, there are ongoing discussions regarding other use cases, possibly out of the intended scope. For example, there are still ongoing discussions on the concept of *automatic modules* and how modules should be *named*. Despite the back and forth from expert groups and industry leaders relating to missing features versus scoped requirements, I trust the Java community will come to a (well-intended) consensus on a robust modular system that will satisfy most use cases out in the wild. To find out more on the topic of Java 9 Jigsaw, I have listed JSRs, books, and links in the appendix of this book.

As a general disclaimer, especially relating to the new Java 9’s platform module system, I will caution you with the following.

■ Disclaimer By the time you read this book, Java 9’s module system will be released. Since Java 9’s module system is new, the language specifications, APIs, terminologies, and concepts are subject to change. This chapter was written using Java 9’s early access edition.

Here is a list of sections to follow when reading this chapter on Java 9’s module system. If you already know the background about Java 9’s module system and its history, you can skip right to the section called “Getting Started.”

- What is Project Jigsaw?
- History (pre-Java 9)
- Getting Started
- Conclusion

What Is Project Jigsaw?

Java modularity, aka “Project Jigsaw,” was an effort to enable developers to partition and encapsulate code in a more manageable manner on the Java platform. By way of partitioning code into modules, application code will be more loosely coupled instead one monolithic application. With good encapsulation, modules can be defined to expose (export) APIs to other modules safely. Java’s new module system will also be backward compatible to code and libraries prior to Java 9 (non-modular code). Not only are partitioning and encapsulating code very good principles, there are other additional benefits, as you will see later.

According to Mark Reinhold, he summarizes JSR 376 (Project Jigsaw):

The overall goal of this JSR is to define an approachable yet scalable module system for the Java Platform. It will be approachable, i.e., easy to learn and easy to use, so that developers can use it to construct and maintain libraries and large applications for both the Java SE and Java EE Platforms. It will be scalable so that it can be used to modularize the Java SE Platform itself, and its implementations.

Mark Reinhold Chief Architect of the Java Platform Group at Oracle, April 1, 2015

Benefits

According to the official web site of project Jigsaw (<http://openjdk.java.net/projects/jigsaw>) the following are the primary goals and benefits of a modular system on the Java platform:

- Make the Java SE platform, and the JDK, more easily scalable down to small computing devices
- Improve the security and maintainability of Java SE platform implementations in general, and the JDK in particular
- Enable improved application performance
- Make it easier for developers to construct and maintain libraries and large applications, for both the Java SE and EE platforms

To achieve these goals, we propose to design and implement a standard module system for the Java SE platform and to apply that system to the platform itself, and to the JDK. The module system should be powerful enough to modularize the JDK and other large legacy code bases, yet still be approachable by all developers.

Project Jigsaw site December 21, 2016

As mentioned relating to partitioning applications these benefits can also lead to scaling down applications to fit into smaller computing devices such as ARM based computers. Having modular code will also help the maintenance and security on the Java SE and Java EE platforms as *microservices* becomes the norm when developing on cloud and container-based application environments.

Since modules make way for reliable configuration building, applications would occupy smaller footprints, thus increasing the performance in areas relating to memory and load times. An example of this is a modular JavaFX application’s start-up speed boost due to a smaller set of modules loaded. Prior to Java 9, applications were developed and packaged with the entire Java runtime. These apps had to load many unused libraries such as *corba* or *sql* packages.

Additional benefits when modularizing Java code (Java 9) are listed as Java Enhancement Proposals:

- Modular runtime Images (JEP 220 at <http://openjdk.java.net/jeps/220>)
- A Java linker for modules (JEP 282 <http://openjdk.java.net/jeps/282>)
- Ahead of time compilation (JEP 295 at <http://openjdk.java.net/jeps/295>)

The JEP 220 is the enhancement proposal for modular runtime images for the Java SE platform. The architects had restructured the entire Java runtime platform such as its directories and core libraries. Restructuring the runtime platform will help improve security, performance, and manageability. For example, libraries such as `rt.jar`, `tools.jar`, and `jfxrt.jar` will no longer be accessible, but are all broken out in modules.

New in Java 9 is *jlink*, the Java linker (JEP 282). *jlink* is a tool to assemble and build applications by linking modules and their dependencies into a custom runtime image. To create custom runtime images, the Java linker is used in conjunction with how the runtime is structured (JEP 220). In other words, the two JEPs are essentially in lock step with each other.

Last of the 3 JEPs mentioned is JEP 295, ahead of time compilation. It's better known as AOT compilation. Although it is considered experimental, the AOT compiler is able to convert Java code into native code on the Linux operating system. This has a nice advantage when environments don't have a Java runtime installed. Another advantage is speed. Since code is statically compiled, there is no need to have JIT (just in time) compilation or warmups. This capability will be used in small devices or highly efficient servers (network devices). By the time you read this, AOT compilation will be supported on popular operating systems.

With these three important Java enhancement proposals working together, you'll begin to see Java-based applications built to be simpler, thinner, faster, and native.

Drawbacks

Although there are many benefits to designing modular code, there are some drawbacks. Some drawbacks are the following:

- Additional complexity when defining modules and dependencies.
- No standard project structure for multiple modules, as it relates to various build tools and IDEs.
- When mixing legacy non-modular code with modular code, the classpath and module path are in play. Debugging code and resolving dependencies may be more difficult.
- Legacy dependencies may never undergo a rewrite to become a Java 9 module, which could risk backward or forward compatibility.

Java 9 Migration Path

Now that you've read about the benefits and drawbacks of the new Java platform module system, you might still be on the fence, because of unknown risks. Perhaps you believe it's still too new. To ease any potential anxiety, Oracle has created an excellent guide to help you migrate legacy Java applications to Java 9. Go for it!

The migration guide is located here: <https://docs.oracle.com/javase/9/migrate>. Also, you should look at important changes in JDK 9 here: <https://docs.oracle.com/javase/9/whatsnew/toc.htm>.

The guide is a very detailed explanation of what changed in Java 9 and how to use platform modules and third-party libraries. The migration guide also talks about an analysis tool called `jdeps`.

The jdeps Analysis Tool

When running `jdeps` against your application code, the tool will inspect code to determine if it relies on deprecated dependencies or JDK internal APIs. For instance, the analysis tool will suggest APIs that should be replaced. Some suggestions offer a replacement for code using packages starting with `com.sun.*`, `sun.misc.*`, etc. For example, if you created a Java application *unmodularized* while referencing a non-public package such as `com.sun.*`, the `jdeps` tool can offer an alternative module to use.

Here is the output after invoking the `jdeps` tool against a Java 8 compiled class file.

```
$ jdeps -cp . com/jfxbe/helloworld/HelloWorld.class
HelloWorld.class -> java.base
HelloWorld.class -> javafx.base
HelloWorld.class -> javafx.controls
HelloWorld.class -> javafx.graphics
HelloWorld.class -> jdk.httpserver
  com.jfxbe.helloworld -> com.sun.net.httpserver jdk.httpserver
  com.jfxbe.helloworld -> java.lang           java.base
  com.jfxbe.helloworld -> javafx.application   javafx.graphics
  com.jfxbe.helloworld -> javafx.collections    javafx.base
  com.jfxbe.helloworld -> javafx.event          javafx.base
  com.jfxbe.helloworld -> javafx.scene          javafx.graphics
  com.jfxbe.helloworld -> javafx.scene.control   javafx.controls
  com.jfxbe.helloworld -> javafx.stage          javafx.graphics
```

The output of the analysis shows the summary and details of platform modules used by the `HelloWorld.class` file. There you will notice the `com.sun.net.httpserver` can be found in the `jdk.httpserver` module. Assuming your application depends on the class `com.sun.net.httpserver.HttpServer`, the suggestion to migrate toward Java modules means you would need to create a `module-info.java` file to require the `jdk.httpserver` module as a dependency.

Let's take a look at another example of using the `jdeps` tool against a Java 9 modularized application JAR file.

The following is an example of analyzing a modularized JAR application by displaying its dependency information.

```
$ jdeps -v helloworld.jar
```

- `-summary` or `-s` Prints the dependency summary.
- `-verbose` or `-v` Prints all classes-level dependencies.

Here's the output after running `jdeps`:

```
$ jdeps -v mlib/com.jfxbe.helloworld.jar
com.jfxbe.helloworld
[file:///Users/cpdea/JavaFX9byExample/Code/jfx9be/chap02/helloworld/mlib/com.jfxbe.
helloworld.jar]
  requires mandated java.base (@9-ea)
  requires javafx.base (@9-ea)
  requires javafx.controls (@9-ea)
  requires javafx.graphics (@9-ea)
  requires jdk.httpserver (@9-ea)
```

```

com.jfxbe.helloworld -> java.base
com.jfxbe.helloworld -> javafx.base
com.jfxbe.helloworld -> javafx.controls
com.jfxbe.helloworld -> javafx.graphics
com.jfxbe.helloworld -> jdk.httpserver
    com.jfxbe.helloworld.HelloWorld -> com.jfxbe.helloworld.HelloWorld$1 com.jfxbe.helloworld
    com.jfxbe.helloworld.HelloWorld -> com.sun.net.httpserver.HttpServer jdk.httpserver
    com.jfxbe.helloworld.HelloWorld -> java.lang.Object java.base
    com.jfxbe.helloworld.HelloWorld -> java.lang.String java.base
    com.jfxbe.helloworld.HelloWorld -> java.lang.Throwable java.base
    com.jfxbe.helloworld.HelloWorld -> javafx.application.Application javafx.graphics
    com.jfxbe.helloworld.HelloWorld -> javafx.collections.ObservableList javafx.base
    com.jfxbe.helloworld.HelloWorld -> javafx.event.EventHandler javafx.base
    com.jfxbe.helloworld.HelloWorld -> javafx.scene.Group javafx.graphics
    com.jfxbe.helloworld.HelloWorld -> javafx.scene.Parent javafx.graphics
    com.jfxbe.helloworld.HelloWorld -> javafx.scene.Scene javafx.graphics
    com.jfxbe.helloworld.HelloWorld -> javafx.scene.control.Button javafx.controls
    com.jfxbe.helloworld.HelloWorld -> javafx.stage.Stage javafx.graphics
    com.jfxbe.helloworld.HelloWorld$1 -> com.jfxbe.helloworld.HelloWorld com.jfxbe.helloworld
    com.jfxbe.helloworld.HelloWorld$1 -> java.io.PrintStream java.base
    com.jfxbe.helloworld.HelloWorld$1 -> java.lang.Object java.base
    com.jfxbe.helloworld.HelloWorld$1 -> java.lang.String java.base
    com.jfxbe.helloworld.HelloWorld$1 -> java.lang.System java.base
    com.jfxbe.helloworld.HelloWorld$1 -> javafx.event.ActionEvent javafx.base
    com.jfxbe.helloworld.HelloWorld$1 -> javafx.event.Event javafx.base
    com.jfxbe.helloworld.HelloWorld$1 -> javafx.event.EventHandler javafx.base
    com.jfxbe.helloworld.HelloWorld$1 -> javafx.stage.Stage javafx.graphics

```

After running `jdeps` against your code, the tool will inform you of internal APIs if any. As you can see in the output, the class `HelloWorld` depends on the following internal API:

`com.sun.net.httpserver.HttpServer`.

Another thing to note regarding the source code in this book is that you'll find some of the source code is in Java 8 or Java 9. Having said this, you will be happy to know that the new Java 9 compiler is able to compile Java 8 source code as is (unmodularized). The module path classes loaded on the classpath are placed into an unnamed module with all public packages exported (read accessible).

Note Some chapters use Java 8 source code from the prior edition of the book, while some chapters use Java 9 source code (modules). There is no need to panic, as source code prior to Java 9 JDK will simply compile and run. In other words, if you encounter Java 8 source code, just treat it as a regular (legacy) Java code or unmodularized code.

Kill Switch

There are cases where legacy code that has been migrated to Java 9 (modularized code) could get a runtime exception raised of type `IllegalAccessException`. This is due to the fact that the code is using `reflection` APIs and Jigsaw's strong encapsulation.

The engineers at Oracle have decided to provide a *kill switch* to allow things to bypass the illegal access check when using reflection. The intent is to give developers more time to migrate code to Java 9 modules; however, the kill switch will be removed in Java 10. The following is an example of using the kill switch to bypass the runtime exception being raised.

```
$ java --permit-illegal-access -jar xyz.jar
```

You can see the tech note that fully explains this at:

<http://mail.openjdk.java.net/pipermail/jigsaw-dev/2017-March/011763.html>

Assuming you are convinced and are ready to jump on the Java Jigsaw train, let's continue the discussion on modules from *past* to *present*. Next, I will elaborate on the history of Java JAR files used as modules. If you find the history of Java modules (prior to Java 9) a little boring, just skip past to the "Getting Started" section.

History

So, modular application and dependency management isn't new? Right?!

In the beginning, Java developers created libraries that consisted of class and resource files zipped into a single Java archive file called a JAR. Each JAR file was considered a *module* within a Java application. As each JAR file was loaded from the classpath, any public APIs within packages could be accessed from the library. Although it was convenient to have accesses to various APIs, things began to get out of hand, which coined the infamous term *JAR hell*. (Actually, it was originally derived from the term *DLL hell*, which developers from the Microsoft operating system developing with C/C++ coined because there existed dynamically linked library files (*.dll) that had a similar issue when resolving dependencies with shared libraries.)

JAR Hell

This term JAR hell is often referred to the JVM's classpath type resolution or dependency resolution (on a single classloader). For example, if you had two JAR files both containing a `Person` class, but both having slightly different methods or attributes (two different versions), you may be wondering which one would be used when instantiating a new `Person()` class?

Of course, this depends on the class loading order, but by default the JVM will load classes in the order it which they were specified on the classpath, such as the `-cp` switch on the command line. Typically, the first one loaded will be the one used (shadowing the second). Some web application environments can enforce load orders, such as parent first or child first. You can see how confusing this can be when you are trying to debug your application and you have no idea which `Person` class it's using.

Because of these known issues, many Java experts formed the OSGi Alliance to create a standard specification as a first step toward developing modular applications on the Java platform. The next section briefly introduces the Java OSGi module system.

OSGi

As Java applications became larger with apps having numerous dependencies, things quickly became a nightmare to manage. To solve Java modularity since 2000 the OSGi Alliance was formed to create an open standards specification for a Java-based modular and services oriented system. OSGi's goal was to provide a specification for vendor's to create implementations of an application container. An application container contains JAR files, called *bundles*, that can be dynamically loaded without restarting the JVM.

In a nutshell, each JAR bundle contains metadata in a `META-INF/manifest.mf` file describing other bundle dependencies. Based on the specification, there are many other ways to export packages and encapsulate bundles. Since each bundle (module) is loaded in a separate class loader, different versions of the same Java class can co-exist on the same JVM. For example, module A depends on the module `com.foo.bar-1.0.jar` while module B depends on the module `com.foo.bar-2.0.jar`, as shown in Figure 2-1. Both JAR files are bundles with each containing their own module definitions (`manifest.mf`).

OSGi Container

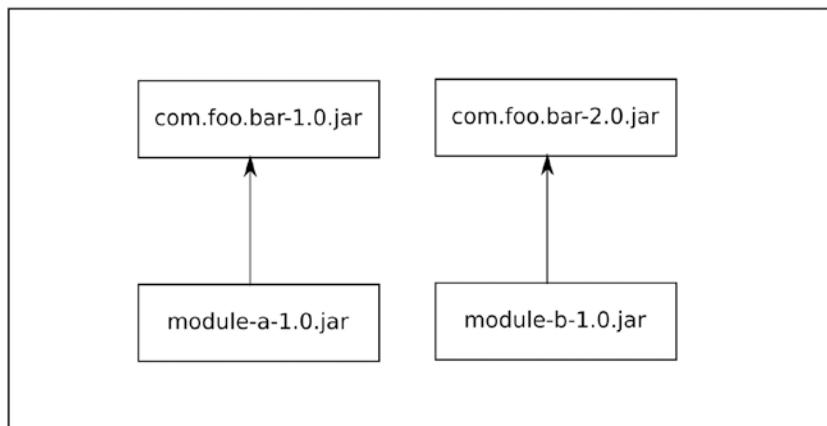


Figure 2-1. An OSGi container with four distinct bundles. Here, module A depends on module `com.foo.bar` version 1.0 and module B depends on module `com.foo.bar` version 2.0.

Assuming the container already contains the two versions of `com.foo.bar`'s JAR bundles, a developer would define module A's manifest, as shown in Listing 2-1, and module B's manifest, as shown in Listing 2-2.

Listing 2-1. Module A's manifest.mf File Containing an OSGi Bundle Definition. Module A Depends on a Package `com.foo.bar` from the JAR File `com.foo.bar-1.0.jar`.

```

Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: module-a
Bundle-SymbolicName: com.carlfxmyservice1
Bundle-Version: 1.0.0
Bundle-Activator: com.carlfxmyservice1.Activator
Import-Package: com.foo.bar;version="1.0"
Export-Package: com.carlfxmyservice1.api;version="1.0.0"

```

Assuming the application is running on an OSGi container, each bundle will undergo a bundle loading lifecycle. Similar to operating systems, where each application has a process ID (PID), an OSGi container will assign each bundle a unique bundle ID. This unique ID allows developers to monitor, stop, and start bundles dynamically.

Listing 2-2. Module B's manifest.mf File Containing an OSGi Bundle Definition. Module B Depends on a Package com.foo.bar from the JAR File com.foo.bar-2.0.jar.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: module-b
Bundle-SymbolicName: com.carlfxmyservice2
Bundle-Version: 1.0.0
Bundle-Activator: com.carlfxmyservice2.Activator
Import-Package: com.foo.bar;version="2.0"
Export-Package: com.carlfxmyservice2.api;version="1.0.0"
```

The OSGi specification spells out all the attributes and naming conventions on how to define manifest files; however, this can be extremely tedious to do by hand. To easily build bundles, check out Bnd tools at <http://bndtools.org>. Also, there are Maven plugins and Gradle plugins for using Bnd tools to assemble bundles.

To be honest, in my experiences with OSGi, I know and believe it is a sound architecture; however, I sense the rate of adoption is low due to its steep learning curve when developing, debugging, and deploying Java-based applications that run on top of it. For example, building HelloWorld on an OSGi container is just overkill. But if you were interested in building an application launcher that managed dynamic services or child apps similar to the Android OS on a mobile device, you may choose to use OSGi. In short, OSGi containers allow you to install services and apps (bundles) without having to reboot the main Java application and JVM.

The following are the top four OSGi implementations available:

- Apache Felix: <http://felix.apache.org>
- Eclipse Equinox: <http://www.eclipse.org/equinox>
- Eclipse Concierge: <http://www.eclipse.org/concierge>
- ProSys: <https://www.bosch-si.com/iot-platform/iot-platform/gateway/software.html>

Maven/Gradle

Another important thing I've not yet mentioned is dependency management when building applications. In other words, when you build a Java application, you will need to obtain the correctly versioned JAR library dependencies in order to compile, deploy, and run your application. Whenever you have one JAR depending on another JAR and so on, thus creating a dependency graph (tree), this is called *transitive dependencies*. Having extremely large transitive dependencies often frustrates developers when it comes time to *trace* and *debug* code. As you will learn later with Java 9's Jigsaw regarding module definitions, you will be able to specify how to map dependencies quite easily. This section briefly mentions the popular build tool Maven.

Repositories

To avoid more JAR hell and manage dependencies properly, the Apache group released Maven, a build automation tool, in 2004. The build tool established conventions for Java projects. One core convention is how a Java project is structured. These project structures also led to a more standard ways to build Java artifacts (modules). While Maven is more than just a build tool, they have also became somewhat of a de facto standard for hosting JARs in repositories. These repositories hold all of the popular open source JAR libraries used today. Although you can have a private repository, the Maven Central is the official public repo on the Internet. I also want to give a shout out to the folks from JFrog (<https://www.jfrog.com/>). JFrog is a company who specializes in build and repository infrastructure services. JFrog's Bintray service allows you to host binaries and executables at their repositories at <https://www.jfrog.com/bintray>.

Maven Coordinates

The folks who created Apache Maven also created the concept called Maven *coordinates*, where modules (*artifacts*) are uniquely named using a group ID, an artifact ID, and a version. For example, Listing 2-3 includes a Maven coordinate to the popular Netty library for network programming. A Maven pom.xml file usually will contain a list of dependencies for compilation.

Listing 2-3. An Excerpt of a pom.xml Containing a Maven Coordinate to a JAR Dependency

```
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-all</artifactId>
    <version>4.0.46.Final</version>
</dependency>
```

To search and locate library dependencies, head over to <https://search.maven.org>. There you will find popular open source libraries. One last mention regarding dependencies and build tools is the Gradle build tool at <https://gradle.org>. Listing 2-4 shows Gradle's way of using Maven coordinates to use dependencies.

Listing 2-4. A Gradle Script with a Dependency Using Maven Coordinates. The Group ID, Artifact ID, and Version Are Separated by Colons.

```
dependencies {
    compile 'io.netty:netty-all:4.0.46.Final'
    testCompile 'junit:junit:4.12'
}
```

Again, I want to remind you that by the time you read this chapter, the build tools such as Maven and Gradle will be able to build (work) against Java 9 module code projects.

Now that you've had a glimpse of the past and current history of Java modules, let's get started using Java 9 modules!

Getting Started

Before you get started implementing modules, this section talks about Java 9's module path. Assuming you already know about the Java classpath, what is new to Java 9 is the *module path* (**--module-path**). Knowing the module path and classpath will help you learn how JAR files and classes get loaded. For example, whenever you are compiling and running code prior to Java 9 with the Java 9 compiler, you'll want to know how Java 9 modules and legacy JAR files co-exist.

What Is the Module Path?

The new Java 9 module path is capable of exposing modules and packages to application code or other modules while hiding packages you don't want exposed. In contrast, the classpath has no ability to encapsulate JARs as modules and their class types. The main issue with the classpath is when you have a public class type from two different JAR files. To see more on the issue, look at the earlier section on history (JAR hell). So, this begs the question, how do you specify the module path?

You'll be shown how to create modules later, but for now you'll learn things a little bit backward. What I mean by this is you'll be shown how to run (execute) a module first and then learn how to create a module.

The module path is a list of elements where each element is either a module JAR file or a module directory. Similar to the classpath, each element is separated by a path separator such as a : or ; for Unix-like OSes and Windows OS, respectively. Here is the usage of the module path when running a module application:

```
$ java --module-path <module path> --module <module name>/<Main class>
```

- **--module-path** or **-p**: A list of modules as JAR files or directories containing modules.
- **--module** or **-m**: The name of the module you want to execute and the main class separated by a forward slash.

Since you will learn how to create and build a modular JAR file later, you'll assume a module already exists on the filesystem. For example, this is how you would specify the module path in order to run an application:

```
$ java --module-path com.jfxbe.physicsengine.jar:mods -m com.jfxbe.game/com.carlfx.game.Main
```

In the example, the first element specified on the **--module-path** is a modularized JAR file named `com.jfxbe.physicsengine.jar`. Here, the JAR file is located local to the current path. The second element specified on the module path is some directory named `mods`. The directory `mods` can be named anything, but as a convention it's short for "modules". The `mods` directory is where the compiled module code goes. The `mods` directory can contain one-to-many module directories. These module directories contain compiled code similar to the classpath. Listing 2-5 is a directory structure of the compiled code in the `com.jfxbe.game` module.

Listing 2-5. A Directory Structure of the Compiled Code in the `com.jfxbe.game` Module

```
mods/
|--- com.jfxbe.game/
|   |--- module-info.class
|   |--- com/
|   |   |--- jfxbe/
|   |   |   |--- game/
|   |   |   |   |--- Main.class
|   |--- images/
|   |   |--- player1.png
|   |   |--- player2.png
|   |   |--- enemy.png
|   |--- soundfx/
|   |   |--- snore.au
```

The `mods` directory contains a module or directory named `com.jfxbe.game` that contains the compiled code and resources. You'll also notice the `module-info.class` file is shown in bold in the `com.jfxbe.game` directory. A class file is created after compiling a `module-info.java` source file. The `module-info.java` source file is called a module definition. Let's take a look at Java 9's module definition syntax.

Module Definition

In short, a module definition is a configuration file listing module dependencies and package exports. New keywords introduced to Java 9 modules are `requires` and `exports`. The `requires` keyword is used to define which modules you'll be needing (dependent on). The `exports` keyword allows other modules to read its packages. Listing 2-6 shows the code for defining a basic module contained in a `module-info.java` file.

Listing 2-6. A Basic Module Definition Having One Dependency and One Package Exported

```
module <module name> {
    requires [public] <module dependency>;
    exports <package name>;
}
```

As a heads-up regarding the syntax in Listing 2-6, the angle brackets and square brackets are not part of the syntax, but a way to let the reader know what is required and what is optional. In Listing 2-6, the angle brackets `<>` denote that it's required and the square brackets mean it's optional. For example, the module name is required and the `public` keyword to the right of `requires` is optional. Let's break things down and explain each item in order to define a module.

Module Naming

Beginning with Listing 2-6 is the `<module name>`. You're probably wondering how to name a module. I know, in particular, I struggled with what to name a module. Actually, it's pretty simple, according to the Jigsaw project specification document on naming module names. It states the following:

Module names, like package names, must not conflict. The recommended way to name a module is to use the reverse-domain-name pattern that has long been recommended for naming packages. The name of a module will, therefore, often be a prefix of the names of its exported packages, but this relationship is not mandatory.

The State of the Module System, Mark Reinhold March 3, 2016

<http://openjdk.java.net/projects/jigsaw/spec/sotms/>

An example of a module using the reverse-domain-name pattern would be the following:

A hypothetical domain name of your company: <http://www.mycompany.com>

The module name should read: `com.mycompany.mymodule`

Requires

Continuing with the basic module definition shown in Listing 2-6, the keyword "requires" references module dependencies. The keyword "requires" indicates that the current module depends on the other named module. One important point to acknowledge is that all modules implicitly depend on (`requires`) the base platform module `java.base`. Also, if you are using JavaFX in your modules, you will implicitly get the module `javafx.base` loaded.

Requires Public (Implied Readability)

In Listing 2-6, the optional [public] modifier for a requires module dependency allows derived modules to have read access implicitly, better known as *implied readability*. An example would be module-A depends on the module java.logging having the public modifier, so another module, let's say module-B, depends on module-A.

Knowing that the logging module is declared public in module-A, module-B will automatically be able to have readability from the module java.logging. This is nice because module-B's module definition wouldn't have to add a statement like requires java.logging. module-B will implicitly get java.logging module for free.

Exports

When using the exports in your module definition, you are exposing package namespaces with public classes.

Although there are other keywords that define other *readability relationship options*, I won't go into those because they are beyond the scope of this book. Suffice it to say, most of the time you will be using the basic module definition shown in Listing 2-6. Now that you know how to run and define modules, you will want to understand module types. Understanding module types will help you know how legacy code and Java 9 modules are loaded.

Module Types

There are four types of modules that the Java runtime will load classes into. Module types are based on whether classes or JARs are used on the classpath or the module path. As modules and classes get loaded into the JVM (class loader), they become one of the four module types. Each module type will have readability access to certain packages and classes. For example, when defining an application module, the implementer will want to expose APIs by using the keyword export to allow packages to be read by other modules and classes. Figure 2-2 shows a typical dependency graph. This dependency graph shows an application module that depends on two platform modules—the unnamed module and two automatic modules, respectively.

Module Types

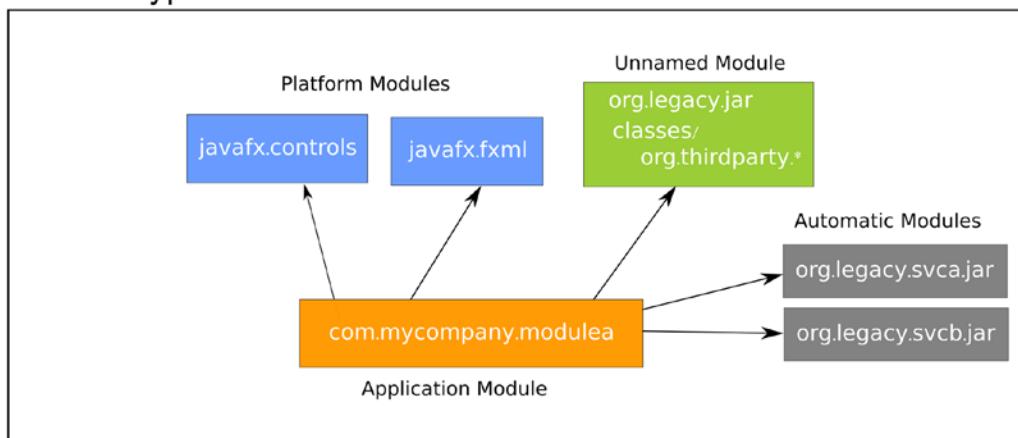


Figure 2-2. A dependency graph of an application module (com.mycompany.modulea) that depends on two platform modules—the unnamed module and two automatic modules

Let's look at the four module types in more detail in order to understand the readability relationships between them.

The following are the descriptions of four module types with their readability access:

- ***Application module***: Created named modules that can read the following module types:
 - Platform
 - Application
 - Automatic
- ***Platform module***: Core modules from the Java runtime cannot read other module types. The core modules can have public access to read other core modules as long as they are defined as `public`, such as `java.logging` or `java.base`.
- ***Automatic module***: JAR libraries that get automatically converted to application module types can read the following module types:
 - Platform
 - Application
 - Automatic
 - Unnamed
- ***Unnamed module***: Any classes or JAR libraries loaded on the classpath will be loaded into the unnamed module. The unnamed module type can read from the following module types:
 - Platform
 - Application
 - Automatic

The unnamed module can't read another *unnamed module* because there is only one (per classloader). JARs and classes loaded from the classpath get loaded into the *unnamed module*. All the public class types and packages are exported to allow classes with the unnamed module to access other classes.

Application Module (Explicit Modules)

Application modules are the modules you create as libraries or applications. As a reminder on a naming convention for modules, you'll want to use the *reverse-domain-name* pattern such as the following:

```
module com.mycompany.modulea {
    // rest of the declaration
}
```

When running your module application, you will need to specify it on the module path, as shown here.

```
$ java --module-path mods/com.mycompany.modulea ...
```

You'll see the `mods` directory, which is created to hold the compiled modules.

Platform Modules (Implicit)

Platform modules are considered *implicit modules* that are core Java and JavaFX platform modules used in application modules. Later you will be creating a `HelloWorld` application using platform modules.

Here is a list of common Java and JavaFX platform modules:

```
java.base
java.logging

javafx.base
javafx.controls
javafx.fxml
javafx.graphics
javafx.media
javafx.swing
javafx.web
```

JavaFX Platform Module Graph

Not much has changed between JavaFX 8 and JavaFX 9; however, JavaFX 9 is newly *modularized*. In other words, JavaFX classes have been broken out into platform modules as opposed to all JavaFX classes put into the `jfxrt.jar` file. When creating modules, you will want to know which JavaFX modules you'll want to depend on (`requires`).

To show JavaFX 9's dependency graph, I recall slides describing the readability relationship from Jonathan Giles' talk on *JavaFX 9 – New & Noteworthy* at JavaOne 2015. Jonathan is a JavaOne Rockstar speaker and JavaFX technical lead at Oracle Corp. To depict the entire JavaFX module graph, Figure 2-3 shows all modules that depend on the `javafx.base` module.

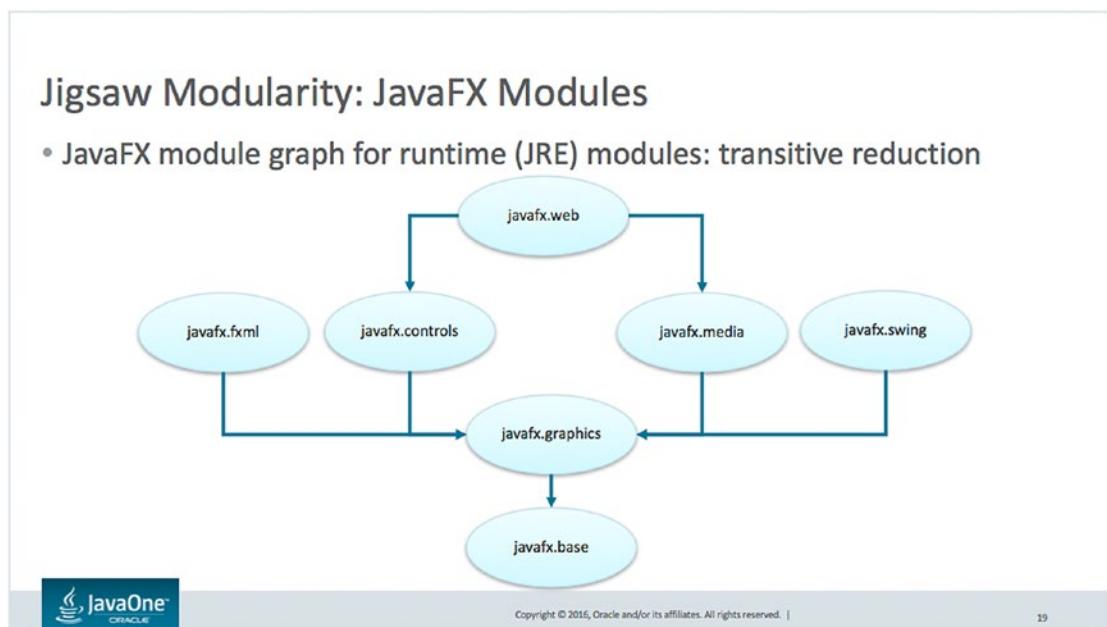


Figure 2-3. The entire JavaFX module graph. All derived modules depend on `javafx.base`

As described, JavaFX platform modules depend on `javafx.base`. Actually, they all do not depend on `javafx.base`, but depend on `javafx.graphics`. The `javafx.graphics` module's definition will have the `public` modifier next to the `javafx.base`, thus making the module `javafx.base` read accessible to derived modules of the `javafx.graphics` module, as shown here:

```
module javafx.graphics {
    requires public javafx.base;
}
```

Also, shown in Figure 2-4 are the other JavaFX platform modules that require the module `javafx.graphics` while having read access implicitly to the `javafx.base` module.

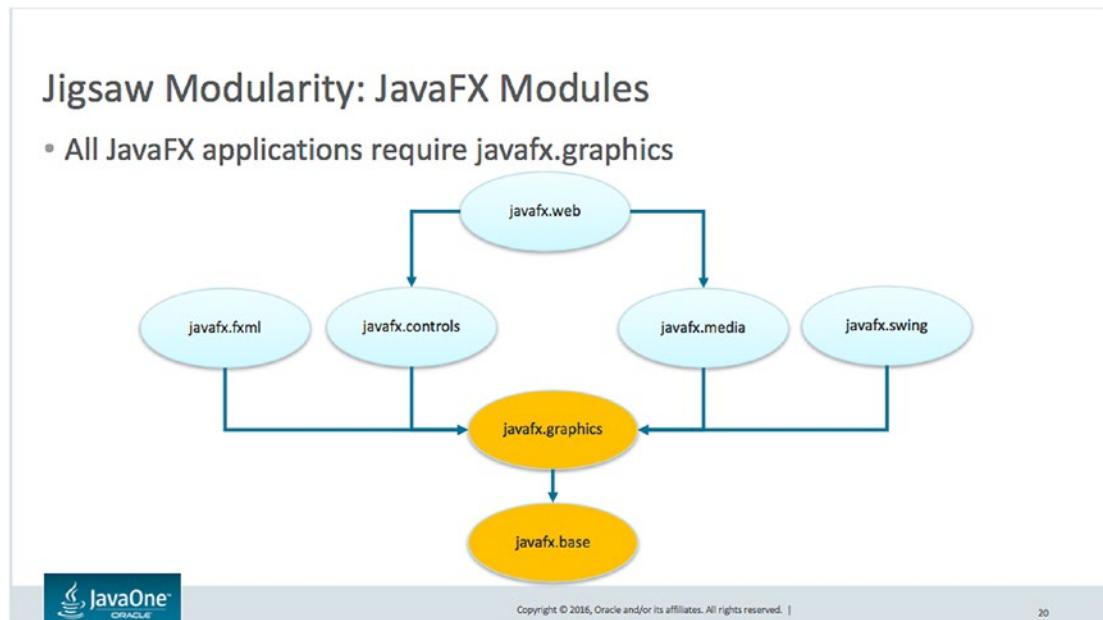
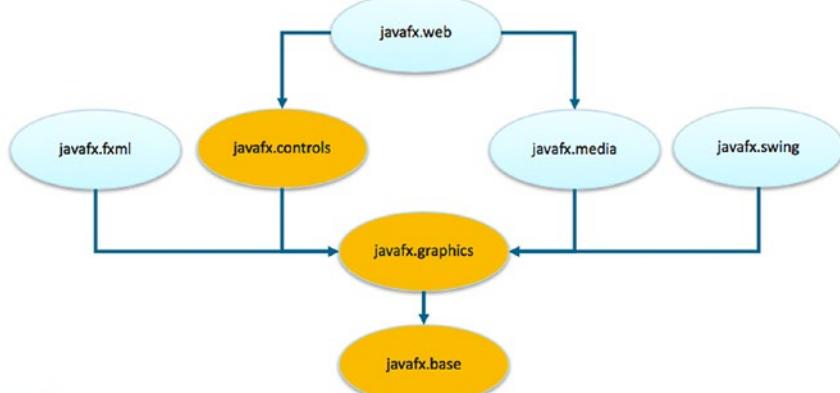


Figure 2-4. *javafx.graphics depends on javafx.base*

Figure 2-5 is the module `javafx.controls` depending on the module `javafx.graphics`, which ultimately depends on the `javafx.base` module. Assuming you know about implicit readability, creating a module that depends on the `javafx.controls` means you won't need to add a `requires javafx.graphics` or `javafx.base` to your module definition.

Jigsaw Modularity: JavaFX Modules

- All JavaFX UI applications require javafx.controls



Copyright © 2016, Oracle and/or its affiliates. All rights reserved. |

21

Figure 2-5. All JavaFX modules depend on the javafx.graphics module

Now that you know about platform modules, let's look at the rest of the module types, such as the *unnamed module*.

Unnamed Module (Classes and Non-Jigsaw Modules on the Classpath)

Any classes or JAR libraries loaded on the classpath will go into an *unnamed module*. Since it is a single module with *no module name* other application modules cannot reference it using the `requires` in their module definitions, hence the name unnamed or no name module. This unnamed module's main purpose is to export any public classes to JVM. For example, if a JAR library was loaded into the unnamed module, all public classes can be used.

Automatic Modules (JAR Files Loaded as Named Modules)

Any legacy JAR file (non-modularized) can be converted to a named module. Instead of legacy JAR files used on the classpath, JARs can be put on the module path to be automatically converted to a named module. The current naming algorithm is in the Javadoc documentation on the `ModuleFinder` class of the `java.base` module.

<http://download.java.net/java/jigsaw/docs/api/java/lang/module/ModuleFinder.html#of-java.nio.file.Path...->

Now that you know about all the different types of modules and how they are loaded, I will show you how to create a JavaFX HelloWorld example application using Java 9 modules.

An Example HelloWorld JavaFX 9 Modular Application

In this example, you will be creating a `HelloWorld` JavaFX 9 modular application. To build a Java 9 project, use the following steps.

Create Project Structure

Create a project directory with the following command-line statements:

```
# Windows
C:\>md \helloworld\src\

# Mac/Linux
$ mkdir -p helloworld/src
```

Create the main module's source directory:

```
# Windows
C:\> cd helloworld
C:\helloworld> md \src\com.jfxbe.helloworld

# Mac/Linux
$ cd helloworld
$ mkdir src/com.jfxbe.helloworld
```

Create the Java package directory structure to hold your source code files.

```
# Windows
C:\helloworld> md \src\com.jfxbe.helloworld\com\jfxbe\helloworld

# Mac/Linux
$ mkdir -p src/com.jfxbe.helloworld/com/jfxbe/helloworld
```

The following directory structure is created for the module application `helloworld`:

```
|---- helloworld/
|---- src/
|---- com.jfxbe.helloworld/
|---- com/
|---- jfxbe/
|---- helloworld/
```

Create a Module Definition

Create a file named `module-info.java` in the `helloworld/src/com.jfxbe.helloworld` directory. Define a `module-info.java` file for the `HelloWorld` module.

```
module com.jfxbe.helloworld {
    requires javafx.controls;
    exports com.jfxbe.helloworld;
}
```

The following is how the directory structure should look:

```
|---- helloworld/
|---- src/
|---- com.jfxbe.helloworld/
|---- module-info.java
|---- com/
|---- jfbe/
|---- helloworld/
```

Next, you will create the JavaFX HelloWorld example code in the helloworld directory.

Create Main Application Code

Create the `HelloWorld.java` file to be placed in the `src/com.jfxbe.helloworld/com/jfbe/helloworld` directory. Just enter the source code from Listing 2-7 into a text edit and save it. As a reminder, the package name is `com.jfxbe.helloworld`.

Listing 2-7. The Main HelloWorld Application

```
package com.jfxbe.helloworld;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;

/**
 * A JavaFX Hello World
 * @author carldea
 */
public class HelloWorld extends Application {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        stage.setTitle("Hello World");
        Group root = new Group();
        Scene scene = new Scene(root, 300, 250);
        Button btn = new Button();
        btn.setLayoutX(100);
```

```

btn.setLayoutY(80);
btn.setText("Hello World");
btn.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        System.out.println("Hello World");
    }
});
root.getChildren().add(btn);
stage.setScene(scene);
stage.show();
}
}

```

Compile Code (Module)

Assume you are at the directory above the `src` directory (`helloworld`) before compiling the source code. Next, you must compile the application as a module.

On the Linux/MacOS X platform, do the following:

```
$ javac -d mods/com.jfxbe.helloworld src/com.jfxbe.helloworld/module-info.java \
src/com.jfxbe.helloworld/com/jfxbe/helloworld/HelloWorld.java
```

On the Windows platform, do the following:

```
C:\helloworld>javac -d mods\com.jfxbe.helloworld src\com.jfxbe.helloworld\module-info.java
src\com.jfxbe.helloworld\com\jfxbe\helloworld\HelloWorld.java
```

If you are going to use third-party modules, you will have to compile your code with the `--modules-path` switch to let the compiler know what additional modules need to be put on the module path. For example, if your `HelloWorld` module needed to use the class `com.xyzcompany.ServiceA` from a third-party Java 9 module named `com.xyzcompany.coolmodule`, you will need to add the entry into the `module-info.java` file and then compile the code like so:

On the Linux/MacOS X platform, do the following:

```
$ javac --modules-path mlibs -d mods/com.jfxbe.helloworld $(find src -name "*.java")
```

On the Windows platform, do the following:

```
C:\helloworld>javac --modules-path mlibs -d mods\com.jfxbe.helloworld src\com.jfxbe.
helloworld\module-info.java src\com.jfxbe.helloworld\com\jfxbe\helloworld\HelloWorld.java
```

The third-party module used in this example resides in the `mlibs` directory as a JAR file.

Copy Resources

Since this `HelloWorld` application does not have any resources such as `.fxml` files, images, or sound files, you can skip this step. But if you have resources to be copied over to the module directory, use the following commands. Change the following command to your satisfaction. Here are the commands to recursively copy files containing the suffix `*.fxml` to the `mods/com.jfxbe.helloworld/com/jfxbe/helloworld` directory.

On the Linux/MacOS X platform, do the following:

```
$ cp -r src/*.fxml mods/com.jfxbe.helloworld/com/jfxbe/helloworld
```

On the Windows platform, do the following:

```
C:\helloworld> xcopy C:\helloworld\src\*.fxml C:\helloworld\mods\com.jfxbe.helloworld /S /Y
```

These commands copy files that end in .fxml recursively and place them in the module directory. This is important when your code tries to invoke the `getResource()` method.

Run Application

Here is how you run the application by specifying the module and class with a main application:

```
$ java --module-path mods -m com.jfxbe.helloworld/com.jfxbe.helloworld.HelloWorld
```

Figure 2-6 shows the output of running the `HelloWorld` JavaFX module application.

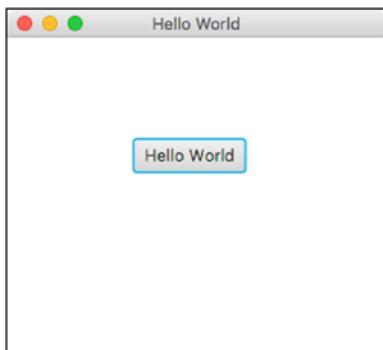


Figure 2-6. Output of the `HelloWorld` JavaFX modularized application

Package Application as JAR

The previous module was compiled as classes that were put into the `mods/com.jfxbe.helloworld` directory. Wouldn't it be nice to create a JAR file executable? Well, let's begin by creating a `mlib` directory. This can be named anything, but, as a convention, the `mlib` stands for module libraries. The `mlib` directory will contain the JAR module application. The following explains how to package a Java 9 modular application.

On the Linux/MacOS X platform, do the following:

```
$ mkdir mlib
$ jar --create --file=mlib/com.jfxbe.helloworld.jar \
--main-class=com.jfxbe.helloworld.HelloWorld -C mods/com.jfxbe.helloworld/.
```

On the Windows platform, do the following:

```
C:\helloworld> md mlib
C:\helloworld> jar --create --file=C:\helloworld\mlib\com.jfxbe.helloworld.jar --main-class=com.jfxbe.helloworld.HelloWorld -C C:\helloworld\mods\com.jfxbe.helloworld.
```

The switches are described as follows:

- **--create:** Creates a JAR file
- **--file:** The JAR file to be created
- **--main-class:** The main class as a JavaFX application
- **-C:** This switch helps change directories and code to turn into

Run Application as JAR

Another way to run the JavaFX application is to run it as a JAR file. Since the JAR was created with a main class specified, the following commands don't include the class name to run the application. This will cut down on the typing.

```
java --module-path mlib -m com.jfxbe.helloworld
```

Since the JAR module definition points to the Java's main class, you will notice that **-m** does not need to specify the package name/class type. This is just a shortcut instead of the more robust way of Java.

Display Module Description

Now that you know how to package your application as a JAR file, you may want to peek into or display the module info with its dependencies. The following command will read and display a module's **module-info** file.

```
jar --describe-module --file=mlib/com.jfxbe.helloworld.jar
```

The following output shows the use of the **--describe-module** switch.

```
module com.jfxbe.helloworld (module-info.class)
  requires mandated java.base
  requires javafx.base
  requires javafx.controls
  requires javafx.graphics
  exports com.jfxbe.helloworld
  main-class com.jfxbe.helloworld.HelloWorld
```

Summary

In this chapter, you learned about Java 9's project Jigsaw to modularize Java code. You got a chance to look at the benefits and drawbacks of modularizing. You ran the `jdeps` tool to assist you in your migration path from JDK 8 to JDK 9. After that, you heard about the term of JAR hell, which basically means Java classes will conflict when different versions are loaded from the classpath. You also learned a little about OSGi containers and how to easily create module bundles. OSGi contains a plethora of attributes as bundle definitions.

Along with modules, you got a glimpse of Maven/Gradle build tools on managing dependencies. These build tools also have repositories that enable developers to store and retrieve artifacts.

Next, you learned how to define a module definition in a `module-info.java` file. After defining a module, you learned about four module types. The following module types are: Platform, Unnamed, Automatic, and Application. You learned about more details regarding the JavaFX platform modules by seeing the entire module graph.

Last but not least, you got a chance to create a JavaFX HelloWorld application as a Java 9 module. You were able to learn how to compile and create a Java 9 JAR module.

Well, there you have it—Java 9 modules.

CHAPTER 3



JavaFX Fundamentals

There is nothing more frustrating than receiving complicated advice as a solution to a problem. Because of this, I have always made it a point to focus on the fundamentals. In the case of JavaFX, you will learn about fundamental drawing primitives that will be used on the drawing surface (JavaFX scene). In order to render graphics on the JavaFX scene, you need basic shapes and colors. Expanding on your knowledge of the JavaFX scene graph, which you visited toward the end of Chapter 1, you will learn in this chapter how to draw and manipulate 2D shapes on the JavaFX scene graph.

Even more fundamental than a shape object is JavaFX's `javafx.scene.Node` class. The Node class is a fundamental base class for all JavaFX scene graph nodes. The Node class provides the ability to transform, translate, and apply effects to any node. Many examples in this chapter involve the `javafx.scene.shape.Shape` class, which is a descendant of the Node class. Therefore, the Shape class will inherit all of the Node class's capabilities. In this chapter, you will explore many derived classes that inherit from Shape. You will begin this chapter by examining JavaFX's most basic shape, the Line node.

JavaFX Lines

Rendering lines in JavaFX is conceptually similar to Euclidean geometry in that two (x, y) coordinate points connect to draw a segment into space. When drawn on the JavaFX scene graph, lines are rendered using the screen coordinate space (system). In geometry, a line is defined as a segment connected by two points in space, although it lacks a width (thickness) and color value. You may be asking the question, "Does this mean that lines are invisible?". In the computer world, physical devices plot pixels and shapes that occupy real surfaces. So, lines drawn on these surfaces will have what is known as a pixel or stroke width. Monitors and printers are examples of such surfaces. Because of this method of production, modern graphics programming has adopted the standard screen coordinate system. In this system, lines that are drawn are visible and have attributes of width and color.

If you remember in grade school in Algebra class, a Cartesian graph is drawn with an X and Y axis where the origin point (0, 0) is where both axes converge at the center. On a computer screen, the coordinate system has the origin (0, 0) at the upper-left corner; this differs from the Cartesian coordinate system, where (0, 0) is placed at the center of the graphing area. In Figure 3-1, it depicts the two coordinate systems as a comparison. The Cartesian coordinate system shown on the left has all four quadrants as visible plotting areas. Shown on the right side in Figure 3-1 is the Screen coordinate system that has the bottom-right quadrant (gray region) as the screen's visible plotting area.

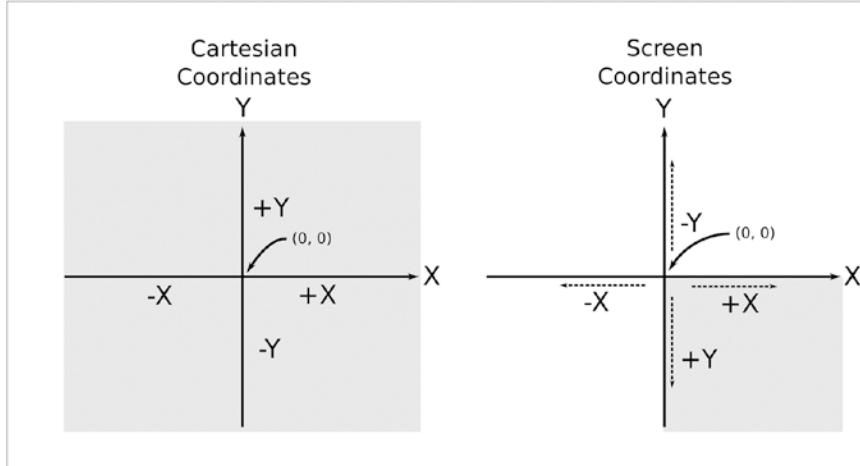


Figure 3-1. *Cartesian versus screen coordinate system. In a Cartesian coordinate system, the origin point (0, 0) is in the center, while a screen coordinate system's origin point (0, 0) is at the upper-left of a screen or monitor.*

Taking a closer look at Figure 3-1, you'll see both coordinate systems. The x coordinate has the same effect when moving a point along the x axis. However, when using the screen coordinate system to move along the y axis, the effect is opposite that of the Cartesian system. In other words, the y coordinate's value increases when moving a point in a downward direction (top to bottom). Figure 3-2 illustrates shapes drawn using the screen coordinate system. Also note that using negative values will plot objects off the screen, such as the star shape partially shown in the viewable area.

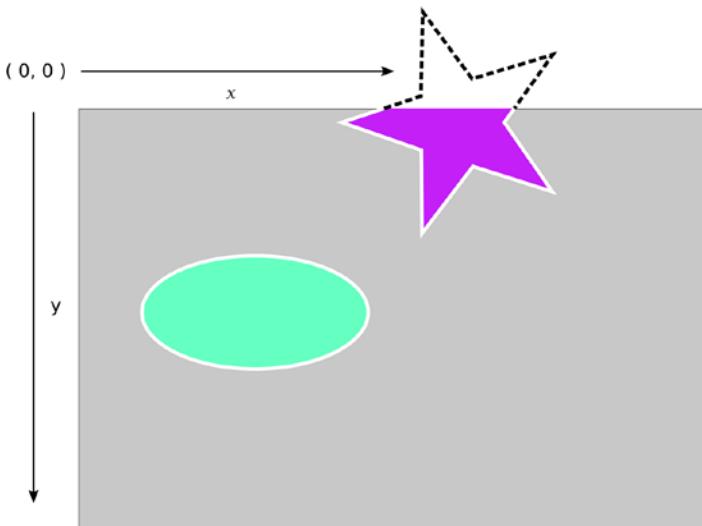


Figure 3-2. *The screen coordinate system*

As you learn more about JavaFX, you will discover many scene graph objects, such as lines, circles, and rectangles. These objects are derived classes of the Shape class. All shape objects can perform geometric operations between two shaped areas, such as subtract, intersect, and union. By mastering the [Shape API](#), you will begin to see endless possibilities. Now let's discuss how to create lines.

To draw lines in JavaFX, you will use the [javafx.scene.shape.Line](#) class. When creating a Line instance, you need to specify a start (x, y) coordinate and an end (x, y) coordinate to draw a line. There are two ways to specify the start and end points when creating Line nodes. The first way is to use a Line's constructor with the parameters startX, startY, endX, and endY. The second way is to instantiate the Line class with the empty constructor and subsequently use its setter methods. Although these two methods of creating a Line node sound pretty basic, I want to point out that there used to be a third method of constructing lines using deprecated *builder classes*. You will see later in this chapter why the JavaFX team decided to deprecate those classes. The data types of all the parameters of a Line class are double type values, which provide floating-point decimal precision. The following code snippet shows the two methods of creating a Line shape node.

```
Line method1 = new Line(startX, startY, endX, endY);

Line method2 = Line();
method.setStartX(startX);
method.setStartY(startY);
method.setEndX(endX);
method.setEndY(endY);
```

Relating to decimal precision values used as screen coordinates, keep in mind that container classes that extend the Region class have a [snapToPixelProperty\(\)](#) method property with a default value of true. Being true, the children nodes will use integer precision as values for screen coordinates. The Javadoc documentation of the Node class describes how pixels are rendered by stating: "At the device pixel level, integer coordinates map onto the corners and *cracks* between the pixels and the centers of the pixels appear at the *midpoints* between integer pixel locations."

Using integer values (whole numbers) by drawing a line or shape with a stroke width of 1 will tend to look *fuzzy*. Based on the Javadoc documentation, the coordinate maps to the upper-left corner (*crack*) to draw the pixel occupying two pixels. For example, if you're drawing a horizontal line having a starting point at (1, 1) and an ending point at (8, 1), the JavaFX coordinates will draw pixels on both sides of the *crack*, as shown in Figure 3-3. The cracks are shown as grid lines. Each grid block (square) is denoted as a pixel. The algorithm also uses varied color intensities.

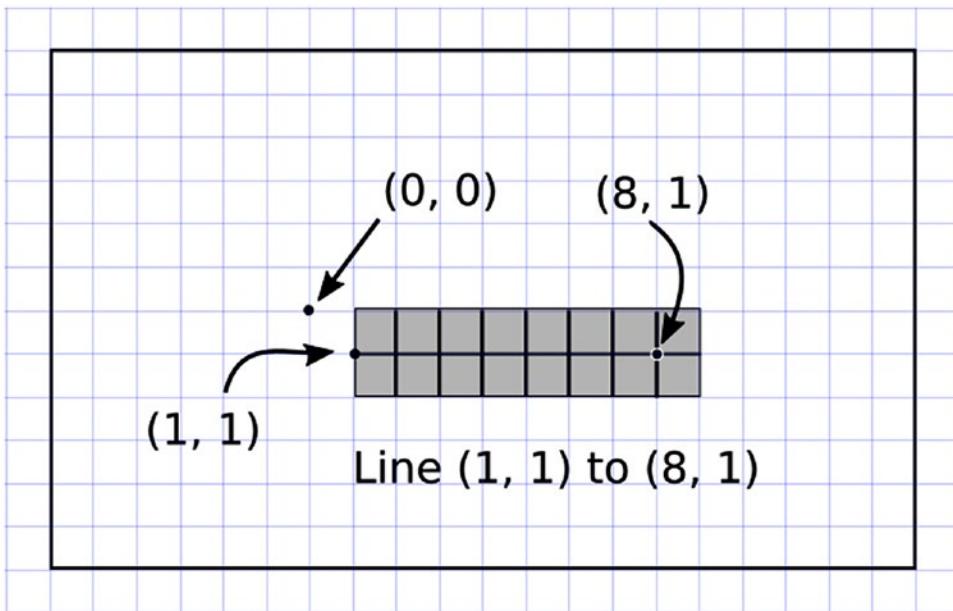


Figure 3-3. A line drawn using integer values will appear thick and fuzzy (blurry). In JavaFX lines using integer values will occupy two pixels. Also, the color of the pixel could change based on the fractional values of the coordinate or the width of the stroke.

To explore JavaFX's further, I created a simple example of various lines drawn on the scene graph, as shown in Figure 3-4. These lines are mostly identical; however, when using integer values for coordinates, lines will tend to look thick or fuzzy. When coordinates have floating point values that have a remainder of less than or equal to .5, the line will appear thin (occupying one pixel). Notice the Thin 2 line as a light gray, making it appear even thinner, yet still occupying one pixel.

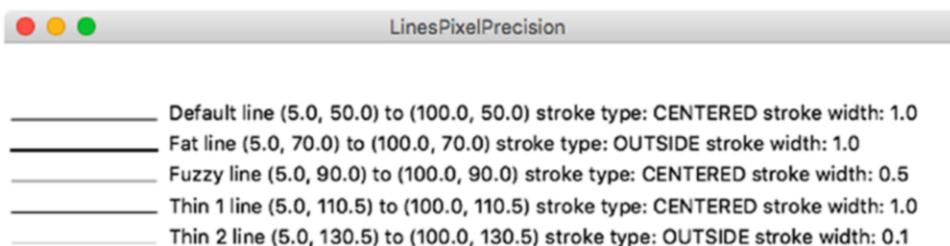


Figure 3-4. Drawn are five lines—Default, Fat, Fuzzy, Thin 1, and Thin 2

Assuming your lines appear fuzzy (using integer coordinate values), you will likely want your lines sharp and thin. To have a line stroke be thin, just add .5 to your coordinate value, which maps to the inner part of the stroke pixel of the screen coordinate point. When using a fractional coordinate value, JavaFX will render the line as one pixel, thus making it look thin. Looking at a line a pixel wide is rather difficult, so Figure 3-5 shows the five lines zoomed in.

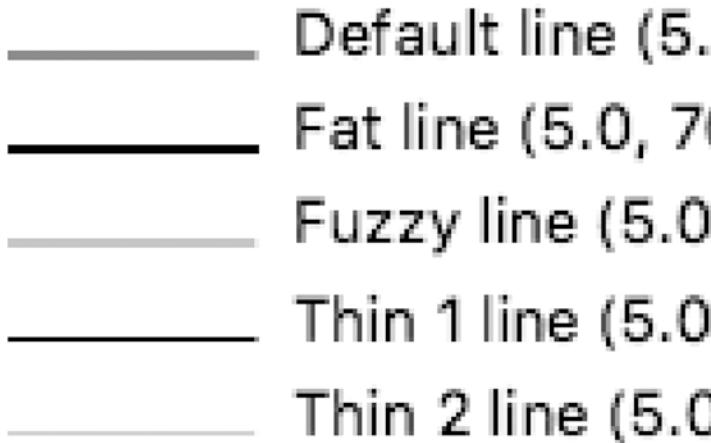


Figure 3-5. A zoomed in picture of Figure 3-4 to examine the pixels drawn for each line. Notice Thin 2 line is occupying one pixel like Thin 1 line, but with a light gray color.

Continuing the discussion of creating lines, the following line has a start point (100.0, 10.0) and an end point (10.0, 110.0) and will be created using a constructor.

```
Line line = new Line(100.0, 10.0, 10.0, 110.0);
```

The second way to create a line node is to instantiate a `Line` class by using the empty default constructor and subsequently setting each attribute using associated setter methods. The following code snippet shows how to specify a line's start and end points using setter methods.

```
Line line = new Line();
line.setStartX(100);
line.setStartY(10);
line.setEndX(10);
line.setEndY(110);
```

Line nodes drawn on the scene graph default to a stroke width of 1.0 (double) and a stroke color of black (`Color.BLACK`). According to the Javadoc, all shapes (such as `Text`, `Rectangle`, etc.) have a stroke color of null (no color) except for `Line`, `Polyline`, and `Path` nodes.

Now that you know how to draw lines on the scene graph, you are probably wondering how to be more creative with lines. Creating different kinds of lines is simple; you basically set properties inherited from the parent class (`javafx.scene.shape.Shape`). Table 3-1 shows the properties you can set on a line (Shape). To retrieve or modify each property, you will use its appropriate getter and setter methods. The table lists each property name and its data type, with a description. Refer to the Javadoc documentation for more details.

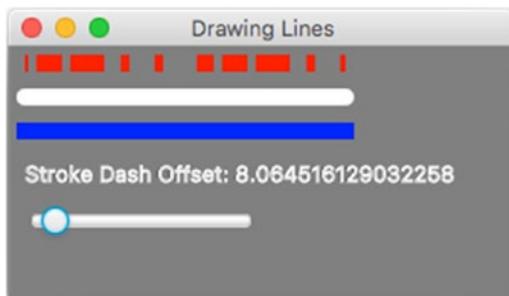
JavaFX provides two ways to style nodes in the scene graph, either programmatically or using CSS styling. For now, you will be learning how to use the JavaFX APIs to style nodes programmatically. Later on in Chapter 15, you will learn how to use JavaFX CSS to create custom controls and how to theme (skin) user interfaces. Next, you will continue learning about JavaFX Lines through an example.

Table 3-1. The `javafx.scene.shape.Shape` Class Properties

Property	Data Type	Description
Fill	<code>javafx.scene.paint.Paint</code>	A color to fill inside a shape.
Smooth	<code>Boolean</code>	True turns on anti-aliasing, otherwise it's false.
<code>strokeDashOffset</code>	<code>Double</code>	The offset (distance) into the dashed pattern.
<code>strokeLineCap</code>	<code>javafx.scene.shape.StrokeLineCap</code>	The cap style on the end of a line or path. There are three styles: <code>StrokeLineCap.BUTT</code> , <code>StrokeLineCap.ROUND</code> , and <code>StrokeLineCap.SQUARE</code> .
<code>strokeLineJoin</code>	<code>javafx.scene.shape.StrokeLineJoin</code>	Decoration when lines meet. There are three types: <code>StrokeLineJoin.MITER</code> , <code>StrokeLineJoin.BEVEL</code> , and <code>StrokeLineJoin.ROUND</code> .
<code>strokeMiterLimit</code>	<code>Double</code>	The limit of a miter joint. Used along with the miter join decoration (<code>StrokeLineJoin.MITER</code>).
<code>stroke</code>	<code>javafx.scene.paint.Paint</code>	A color for the shape's line stroke.
<code>strokeType</code>	<code>javafx.scene.shape.StrokeType</code>	Where to draw the stroke around the boundary of a Shape node. There are three types: <code>StrokeType.CENTERED</code> , <code>StrokeType.INSIDE</code> , and <code>StrokeType.OUTSIDE</code> .
<code>strokeWidth</code>	<code>Double</code>	A stroke line's width.

Drawing Lines

To get a better idea of how you would use a shape's properties based on Table 3-1, let's look at an example. Figure 3-6 is the output of Listing 3-1, the `DrawingLines.java` source code, demonstrating the drawing of JavaFX lines. The JavaFX application in the listing draws three lines with various properties modified. Some common properties used in this example are stroke dash offset, stroke line cap, stroke width, and stroke color.

**Figure 3-6.** Drawing lines

In Figure 3-6, you can see that the first line (top) is a thick red line with a dashed stroke pattern. The second line is a thick white line having rounded end caps (line cap). Last is an ordinary blue line having the same thickness as the others. You will also notice in Figure 3-6 underneath the blue line are two controls—a label and a slider control. These controls allow the user to change the red (top) line's stroke dash offset property dynamically. The label control displays the dash offset value as the slider control moves. Listing 3-1 shows the source code for DrawingLines.java.

Listing 3-1. DrawingLines.java

```
package jfxbe;

import javafx.application.Application;
import javafx.beans.value.ObservableValue;
import javafx.scene.*;
import javafx.scene.control.Slider;
import javafx.scene.paint.Color;
import javafx.scene.shape.*;
import javafx.scene.text.Text;
import javafx.stage.Stage;

/**
 * Drawing Lines
 * @author carldea
 */
public class DrawingLines extends Application {

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Drawing Lines");

        Group root = new Group();
        Scene scene = new Scene(root, 300, 150, Color.GRAY);

        // Red line
        Line redLine = new Line(10, 10, 200, 10);

        // setting common properties
        redLine.setStroke(Color.RED);
        redLine.setStrokeWidth(10);
        redLine.setStrokeLineCap(StrokeLineCap.BUTT);

        // creating a dashed pattern
        redLine.getStrokeDashArray().addAll(10d, 5d, 15d, 5d, 20d);
        redLine.setStrokeDashOffset(0);

        root.getChildren().add(redLine);

        // White line
        Line whiteLine = new Line(10, 30, 200, 30);
        whiteLine.setStroke(Color.WHITE);
        whiteLine.setStrokeWidth(10);
        whiteLine.setStrokeLineCap(StrokeLineCap.ROUND);
```

```

root.getChildren().add(whiteLine);

// Blue line
Line blueLine = new Line(10, 50, 200, 50);
blueLine.setStroke(Color.BLUE);
blueLine.setStrokeWidth(10);

root.getChildren().add(blueLine);

// slider min, max, and current value
Slider slider = new Slider(0, 100, 0);
slider.setLayoutX(10);
slider.setLayoutY(95);

// bind the stroke dash offset property
redLine.strokeDashOffsetProperty()
    .bind(slider.valueProperty());
root.getChildren()
    .add(slider);

Text offsetText = new Text("Stroke Dash Offset: " + slider.getValue());
offsetText.setX(10);
offsetText.setY(80);
offsetText.setStroke(Color.WHITE);

// display stroke dash offset value
slider.valueProperty()
    .addListener((ov, curVal, newVal) ->
        offsetText.setText("Stroke Dash Offset: " + newVal));
root.getChildren().add(offsetText);

primaryStage.setScene(scene);
primaryStage.show();
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    launch(args);
}
}

```

`DrawingLines.java` begins by setting the title of the Stage window using the `setTitle()` method. Then it creates a root node (`javafx.scene.Group`) for the Scene object. If you remember back in Chapter 1, which discussed the `HelloWorld` example, the first node added to the JavaFX Scene graph is called the *root* node. The root node is always a JavaFX container type node such as a Group or BorderPane. All container type nodes (those extending `javafx.scene.Parent`) have a method called `getChildren().add()` that allows any JavaFX node to be added to the scene graph. By default, the Scene object will be filled with a white background; however, because one of our lines is white, the code will set the scene to have a color of gray (`Color.GRAY`). This will allow some contrast in order to see the white line. In the paperback edition of the book images are in grayscale color, but if you use the PDF version of the book, the code creates three lines in red, white, and blue, respectively.

In the first line, the code sets the common properties of a Line node. The common properties are *stroke color*, *stroke width*, and *stroke line cap*. As noted earlier, lines do not have a shape internally, so a line's fill color property is set to null (no color) and the stroke color defaults to black. To set the stroke color, you can use built-in JavaFX colors by using the `javafx.scene.paint.Color` class. For instance, for the color red you use `Color.RED`. There are many ways to specify color, such as using the RGB, HSB, or Web hex values. All three methods also have the ability to specify alpha value (transparency). Later, you will see additional methods for coloring shapes. Refer to the Javadoc to learn more about color (`javafx.scene.paint.Color`). After setting the stroke's outline color, you can set the line's stroke width (thickness) by using the `setStrokeWidth()` method. Shapes also have a stroke line cap property. This property specifies the style of the ends of the line. For instance, specifying the stroke line cap as butt (`StrokeLineCap.BUTT`) will make a flat square end, while a round (`StrokeLineCap.ROUND`) style will appear with a rounded end. The following code snippet sets a line node's common shape properties:

```
// setting common properties
redLine.setStroke(Color.RED);
redLine.setStrokeWidth(10);
redLine.setStrokeLineCap(StrokeLineCap.BUTT);
```

After setting the common properties on the red Line node, the example code creates a dashed pattern. To form a dashed pattern, you would simply add double values to the `getStrokeDashArray().addAll()` method. Each value represents the number of pixels for a dash segment. To set the dash segment array, the first value (10d) is a visible dash segment that's 10 pixels wide. Next is a five-pixel empty segment (not visible). Following that is a visible dash segment 15 pixels wide, and so on. Because the array has an odd number of values, you can see that as the pattern repeats itself. The first value (10d) becomes a 10-pixel empty segment (it's not visible). Following is the code to create a dashed pattern for a Line node:

```
// creating a dashed pattern
redLine.getStrokeDashArray().addAll(10d, 5d, 15d, 5d, 20d);
redLine.setStrokeDashOffset(0);
```

By default, the dash offset property value is 0. To see what happens when the offset property changes, the user can drag the slider thumb to the right to increase the offset. The stroke dash offset is the distance into the current pattern to begin the line drawing.

Because the other two lines are pretty much the same in the way they modify common properties, I will not explain them any further. I trust you now understand the fundamentals of creating lines. One last thing to mention about this example is the Slider control and the way it is wired up using binding.

Notice in Listing 3-1 that after the three lines just discussed, the slider control has handler code that updates a Text node dynamically to display the dash offset value. Also notice the invocation to the `addListener()` method, which contains concise code added as a change listener. This may look odd to you; however, it reflects a new Java 8 feature called *lambda expressions*, which you will learn more about in Chapter 4. The following code snippet creates a change listener using lambdas instead of an anonymous inner class (`ChangeListener`).

```
// display stroke dash offset value
slider.valueProperty()
    .addListener( (ov, oldValue, newValue) ->
        offsetText.setText("Stroke Dash Offset: " + newValue));
```

Learning how to draw lines will help you apply the same knowledge to any Shape node in JavaFX. These important concepts will allow you to create any kind of shape styled the way you see fit. Speaking of shapes, they are discussed next.

Drawing Shapes

If you are familiar with drawing programs that allow users to draw shapes on the canvas, you'll notice a lot of similarities when using the JavaFX's Shape API. The JavaFX Shape API allows you to create many common shapes such as lines, rectangles, and circles. Some of the more complex shapes are Arc, CubicCurve, Ellipse, and QuadCurve. JavaFX also lets you create custom shapes for situations where the predefined stock shapes don't fit the bill. In this section, you will be exploring basic shapes, complex shapes, and custom shapes. I will not demonstrate all of the shapes available in JavaFX, because of space limitations. To see all the available shapes, refer to the Javadoc for details.

As stated earlier, JavaFX has common shapes such as lines, rectangles, and circles. In the section "Drawing Lines," you learned the basic method of changing any shape's stroke color, stroke width, and many other attributes. Knowing these skills will be useful throughout this chapter. Let's begin by getting familiar with the rectangle shape.

Drawing rectangles on the scene graph is quite easy. Just as in geometry class, you specify a width, height, and an (x, y) location (upper-left corner) to position the rectangle on the scene graph. To draw a rectangle in JavaFX, you use the `javafx.scene.shape.Rectangle` class. In addition to common attributes, the `Rectangle` class also implements an *arc width* and *arc height*. This feature will draw rounded corners on a rectangle. Figure 3-7 shows a rounded rectangle, which has both an *arc width* and an *arc height*.

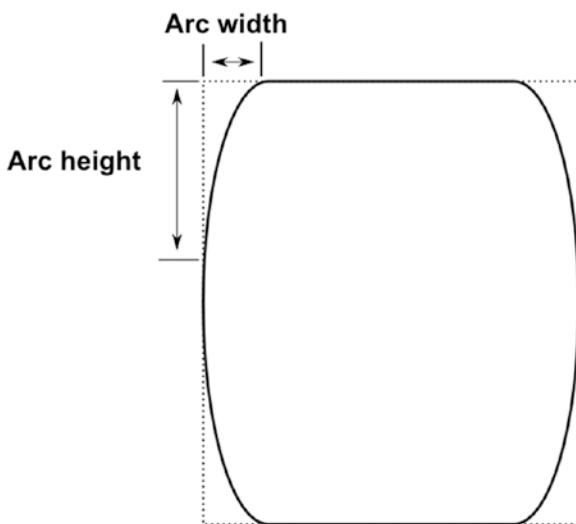


Figure 3-7. A rounded rectangle's arc width and arc height

The following is a code snippet that draws a rectangle positioned at (50, 50) with a width of 100, a height of 130, an arc width of 10, and an arc height of 40.

```
Rectangle roundRect = new Rectangle();
roundRect.setX(50);
roundRect.setY(50);
roundRect.setWidth(100);
roundRect.setHeight(130);
roundRect.setArcWidth(10);
roundRect.setArcHeight(40);
```

Having the basic knowledge of simple shapes such as lines and rectangles will enable you to manipulate other simple shapes having equivalent properties. Armed with this knowledge of simple shapes, let's look at creating more complex shapes.

Drawing Complex Shapes

Learning about simple shapes is great, but to create more complex shapes you need to see what other built-in shapes the JavaFX API has to offer. Exploring the Java documentation (`javafx.scene.shape.*`), you will discover many derived shape classes to choose from. The following are the currently supported shapes:

- Arc
- Circle
- CubicCurve
- Ellipse
- Line
- Path
- Polygon
- Polyline
- QuadCurve
- Rectangle
- SVGPath
- Text (which is considered to be a type of shape)

A Complex Shape Example

To demonstrate drawing complex shapes, the next code example draws four fun cartoon-like shapes. Normally cartoon drawings have a thick stroke width, similar to a penciled-in outline. The first shape in the example is a sine wave, the second an ice cream cone, the third a smile, and the last a donut. Before studying the code, you may want to see the shapes drawn onto the scene, so you can visualize each shape as you look at the code listing. Figure 3-8 is the output of Listing 3-2, depicting four complex shapes.

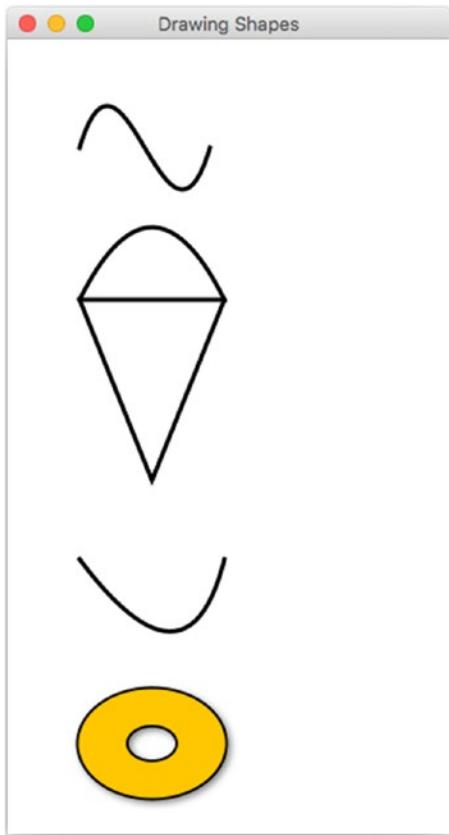


Figure 3-8. Drawing complex shapes

The `DrawingShape.java` code shown in Listing 3-2 demonstrates the drawing of the shapes you see in Figure 3-8. The first complex shape involves a cubic curve (`CubicCurve`) that is drawn in the shape of a sine wave. The next shape is an ice cream cone; it uses the `Path` class, which contains path elements (`javafx.scene.shape.PathElement`). The third shape is a quadratic Bézier curve (`QuadCurve`), forming a smile. The final shape is a delectable donut; to create this donut shape you will create two (`Ellipse`) shapes (one smaller and one larger) then subtracting the smaller shape from the larger shape. For brevity, Listing 3-2 shows just the `start()` method containing the main JavaFX elements. To get the full code listing, download the example code from the book's web site.

Listing 3-2. The Java File `DrawingShape.java` Consists of an Application Capable of Drawing Advanced Shapes

```
@Override  
public void start(Stage primaryStage) {  
    primaryStage.setTitle("Drawing Shapes");  
    Group root = new Group();  
    Scene scene = new Scene(root, 306, 550, Color.WHITE);
```

```
// Sine wave
CubicCurve cubicCurve = new CubicCurve(
    50, // start x point
    75, // start y point
    80, // control x1 point
    -25, // control y1 point
    110, // control x2 point
    175, // control y2 point
    140, // end x point
    75); // end y point
cubicCurve.setStrokeType(StrokeType.CENTERED);
cubicCurve.setStroke(Color.BLACK);
cubicCurve.setStrokeWidth(3);
cubicCurve.setFill(Color.WHITE);

root.getChildren().add(cubicCurve);

// Ice cream cone
Path path = new Path();
path.setStrokeWidth(3);

// create top part beginning on the left
MoveTo moveTo = new MoveTo();
moveTo.setX(50);
moveTo.setY(150);

// curve ice cream (dome)
QuadCurveTo quadCurveTo = new QuadCurveTo();
quadCurveTo.setX(150);
quadCurveTo.setY(150);
quadCurveTo.setControlX(100);
quadCurveTo.setControlY(50);

// cone rim
LineTo lineTo1 = new LineTo();
lineTo1.setX(50);
lineTo1.setY(150);

// left side of cone
LineTo lineTo2 = new LineTo();
lineTo2.setX(100);
lineTo2.setY(275);

// right side of cone
LineTo lineTo3 = new LineTo();
lineTo3.setX(150);
lineTo3.setY(150);
```

```
path.getElements().addAll(moveTo, quadCurveTo, lineTo1, lineTo2, lineTo3);

path.setTranslateY(30);

root.getChildren().add(path);

// A smile
QuadCurve quad = new QuadCurve(
    50, // start x point
    50, // start y point
    125, // control x point
    150, // control y point
    150, // end x point
    50); // end y point
quad.setTranslateY(path.getBoundsInParent().getMaxY());
quad.setStrokeWidth(3);
quad.setStroke(Color.BLACK);
quad.setFill(Color.WHITE);

root.getChildren().add(quad);

// outer donut
Ellipse bigCircle = new Ellipse(
    100, // center x
    100, // center y
    50, // radius x
    75/2); // radius y
bigCircle.setStrokeWidth(3);
bigCircle.setStroke(Color.BLACK);
bigCircle.setFill(Color.WHITE);

// donut hole
Ellipse smallCircle = new Ellipse(
    100, // center x
    100, // center y
    35/2, // radius x
    25/2); // radius y

// make a donut
Shape donut = Path.subtract(bigCircle, smallCircle);
donut.setStrokeWidth(1.8);
donut.setStroke(Color.BLACK);

// orange glaze
donut.setFill(Color.rgb(255, 200, 0));

// add drop shadow
DropShadow dropShadow = new DropShadow(
    5, // radius
    2.0f, // offset X
    2.0f, // offset Y
    Color.rgb(50, 50, 50, .588));
```

```

donut.setEffect(dropShadow);

// move slightly down
donut.setTranslateY(quad.getBoundsInParent().getMinY() + 30);

root.getChildren().add(donut);

primaryStage.setScene(scene);
primaryStage.show();
}

```

Four shapes are drawn in Figure 3-8. Each shape will be detailed further in the following sections, which describe the code and the reasoning behind the creation of each of the four shapes.

The Cubic Curve

In Listing 3-2, the first shape, drawn as a sine wave, is really a `javafx.scene.shape.CubicCurve` class. To create a cubic curve, you simply look for the appropriate constructor to be instantiated. A cubic curve's main parameters to set are `startX`, `startY`, `controlX1` (control point1 X), `controlY1` (control point1 Y), `controlX2` (control point2 X), `controlY2` (control point2 Y), `endX`, and `endY`. Figure 3-9 shows a cubic curve with control points influencing the curve.

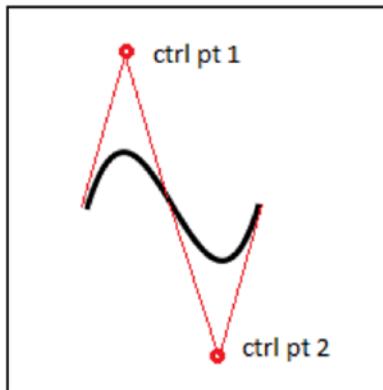


Figure 3-9. Cubic curve

The `startX`, `startY`, `endX`, and `endY` parameters are the starting and ending points of a curved line, and `controlX1`, `controlY1`, `controlX2`, and `controlY2` are control points. The control point (`controlX1`, `controlY1`) is a point in screen space that will influence the line segment between the start point (`startX`, `startY`) and the midpoint of the line. The point (`controlX2`, `controlY2`) is another control point that will influence the line segment between the midpoint of the line and its end point (`endX`, `endY`). A control point is a point that pulls the curve toward the direction of itself. A definition of a control point is a line perpendicular to a tangent line on the curve. This example simply has a control point 1 above the line to pull the curve upward to form a hill and control point 2 below the line to pull the curve downward to form a valley.

Note All older JavaFX 2.x Builder classes are deprecated in JavaFX 8. Be advised that the previous edition of this book used Builder classes. Shape classes using deprecated builder classes should be converted in favor of constructors and setter methods when specifying properties.

At this point, I want to mention that beginning with JavaFX 8, Oracle's JavaFX team has decided to deprecate the many Builder classes in JavaFX. The decision to deprecate builder classes was mainly due to two bugs found in Java 6 and 7 that were fixed in Java 8. The bug fixes would later cause binary compatibility issues that would break the builder classes. To get the full story, see Appendix A under JavaFX 8's features, which references links to the discussions about the issue of builder classes. Knowing this will allow you to recognize older JavaFX 2.x code so that refactoring is painless. The first edition of this book used Builder classes quite frequently, and they are an elegant way to create JavaFX objects. Although the Builder classes are deprecated, it doesn't prevent you from creating APIs with the *builder pattern* in mind. To create objects without an associated builder class, just find the appropriate constructor to be instantiated or use setter methods to set the object's properties.

The Ice Cream Cone

The ice cream cone shape is created using the `javafx.scene.shape.Path` class. Each path element is created and added to the Path object. Also, each element is not considered a graph node (`javafx.scene.Node`). This means that path elements do not extend from the `javafx.scene.shape.Shape` class and they cannot be child nodes in a scene graph. Figure 3-10 shows an ice cream cone shape.

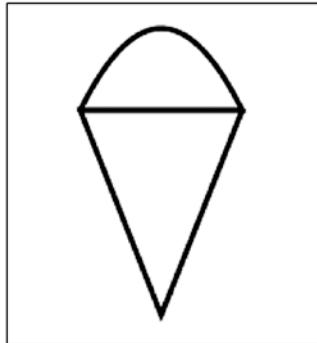


Figure 3-10. The ice cream cone is drawn using JavaFX's Path node

Path elements actually extend from the `javafx.scene.shape.PathElement` class, which is used only in the context of a Path object. So you won't be able to instantiate a `LineTo` class to be placed in the scene graph. Just remember that the classes with To as a suffix are path elements, not Shape nodes.

For example, the `MoveTo` and `LineTo` object instances are `Path` elements added to a `Path` object, not shapes that can be added to the scene. The order in which path elements are added is the order in which things are drawn. The following code snippet are `Path` elements added to a `Path` object to draw an ice cream cone:

```
// Ice cream
Path path = new Path();

MoveTo moveTo = new MoveTo();
moveTo.setX(50);
moveTo.setY(150);

...// Additional Path Elements created.
LineTo lineTo1 = new LineTo();
lineTo1.setX(50);
lineTo1.setY(150);

...// Additional Path Elements created.

path.getElements().addAll(moveTo, quadCurveTo, lineTo1, lineTo2, lineTo3);
```

This snippet of code is repeated from Listing 3-2, although some of the code is omitted for brevity. Figure 3-11 depicts the ice cream cone having the ordered steps in which path elements are drawn.

- Step 1: `moveTo`
- Step 2: `quadCurveTo`
- Step 3: `lineTo1`
- Step 4: `lineTo2`
- Step 5: `lineTo3`

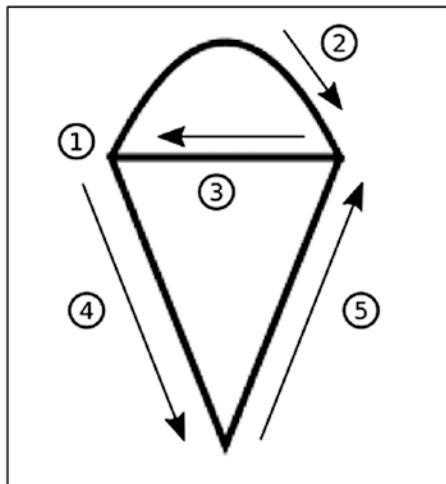


Figure 3-11. The ice cream cone's draw order using Steps 1-5

The Smile

To render the smile shape, the code uses the `javafx.scene.shape.QuadCurve` class. This is similar to the cubic curve example described earlier in the first shape. Instead of two control points, you only have one control point. Again, a control point influences a line by pulling the midpoint toward it. Figure 3-12 shows a `QuadCurve` shape with a control point below its starting and ending points to form a smile.

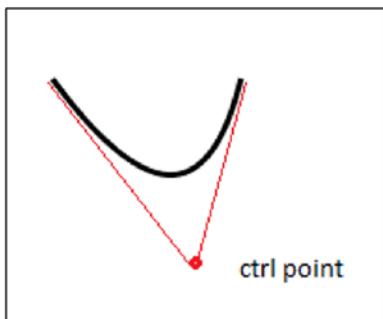


Figure 3-12. Quadratic curve

The following code draws a quadratic curve with a stroke thickness of three pixels filled with the color white:

```
// A smile
QuadCurve quad = new QuadCurve(
    50, // start x point
    50, // start y point
    125, // control x point
    150, // control y point
    150, // end x point
    50); // end y point

quad.setStrokeWidth(3);
quad.setStroke(Color.BLACK);
quad.setFill(Color.WHITE);
```

The Donut

Last is the tasty donut shape with an interesting drop shadow effect shown in Figure 3-13. This custom shape was created using geometric operations, such as subtract, union, intersection, and so on. With any two shapes, you can perform geometric operations to form an entirely new shape object. All of the operations can be found in the `javafx.scene.shape.Path` class.

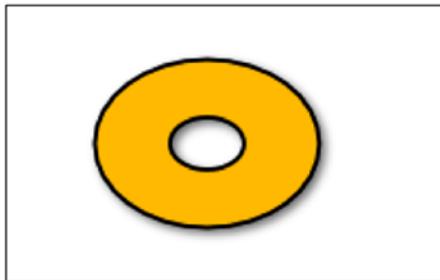


Figure 3-13. A donut shape created by using shape subtraction

To create the donut shape, you begin by creating two circular ellipse (`javafx.scene.shape.Ellipse`) instances. Subtracting the smaller ellipse (donut hole) from the larger ellipse area creates a newly derived Shape object, which is returned using the static `Path.subtract()` method. The following code snippet creates the donut shape using the `Path.subtract()` method:

```
// outer donut
Ellipse bigCircle = ...//Outer shape area

// donut hole
Ellipse smallCircle = ...// Inner shape area

// make a donut
Shape donut = Path.subtract(bigCircle, smallCircle);
```

Next is applying a drop shadow effect to the donut shape. A common technique is to draw the shape filled black while the original shape is laid on top slightly offset to appear as a shadow. However, in JavaFX, the code will draw the donut shape once and use the `setEffect()` method to apply a `DropShadow` object instance. To cast the shadow offset, call the `setOffsetX()` and `setOffsetY()` methods. Typically, if the light source is from the upper-left, the shadow is shown from the lower-right of the shape.

One last thing to point out is that all shapes in the example are initially drawn to be positioned underneath one another. Looking back at Listing 3-2, you'll notice that as each shape was created, its `translateY` property was set to reposition or shift each shape from its original position. For example, if a shape's upper-left bounding box point is created at (100, 100) and you want it to be moved to (101, 101), the `translateX` and `translateY` properties would be set to 1.

As each shape is rendered beneath another in this example, you may invoke the `getBoundsInParent()` method to return the information about the bounding region of a node, such as its width and height. The `getBoundsInParent()` calculation for the height and width includes the node's actual dimensions (width and height) plus any effects, translations, and transformations. For example, a shape that has a drop shadow effect increases its width by including the shadow.

Figure 3-14 is a dashed red rectangle surrounding a `Rectangle` node inside a parent node, better known as the *bounding rectangle in parent* (Bounds in Parent). You will notice that the width and height calculations include transforms, translates, and effects applied to the shape. In Figure 3-14, the transform operation is a rotation and the effect is a drop shadow.

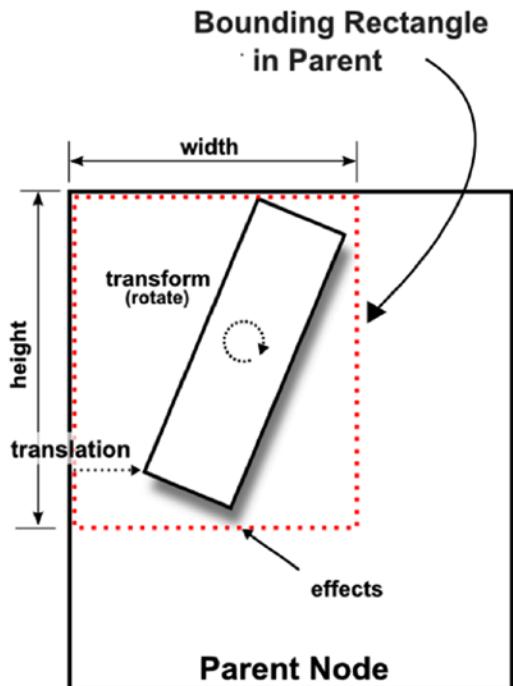


Figure 3-14. Bounding rectangle in parent

Painting Colors

I've mentioned that various drawing programs have their own tools to draw shapes. Drawing programs also provide the ability to paint shapes, using a color palette. Typically, paint programs have a paint bucket tool to fill areas on the canvas. Often, photo editing software packages have the ability to fill using a gradient color or change the lighting of colors. In JavaFX, you can also apply colors (Paint) to objects using similar techniques. In this section, you will see an example of a JavaFX application that showcases three shapes filled with colors (Paint).

An Example of Color

To demonstrate common color fills, Figure 3-15 illustrates an application that displays three shapes with different color fills. It depicts the following three shapes—ellipse, rectangle, and rounded rectangle. Each shape has a gradient fill. The first shape, the ellipse, is filled with a red radial gradient. The second is a rectangle filled with a yellow linear gradient. Finally, the rounded rectangle has a green cycle reflect gradient fill. You'll also notice a thick black line behind the yellow rectangle to illustrate the alpha level (transparency) of colors.

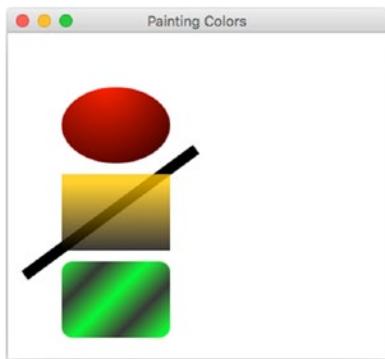


Figure 3-15. Colored shapes

Note The example to follow touches on basic techniques for creating solid colors, gradient colors, and translucent colors. There are also advanced strategies, such as the [ImagePattern API](#) and [Blend API](#). You can read about those, if you are interested, in the API documentation viewable through Javadoc.

In JavaFX, all shapes can be filled with simple colors and gradient colors. As a reminder, according to the Javadoc, all shape nodes are filled with the color black by default, except for the Line, Polyline, and Path classes (descendants of `java.scene.shape.Shape`). Listing 3-3 uses the following main classes that will be used to fill the shape nodes shown in Figure 3-15:

- `javafx.scene.paint.Color`
- `javafx.scene.paint.Stop`
- `javafx.scene.paint.RadialGradient`
- `javafx.scene.paint.LinearGradient`

Note In Figure 3-15, the `blackLine`, `rectangle`, and `roundRectangle` shapes are all positioned by the `setTranslateY()` method relative to the `ellipse` shape and relative to one another.

Listing 3-3. PaintingColors.java

```
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("Painting Colors");
    Group root = new Group();
    Scene scene = new Scene(root, 350, 300, Color.WHITE);

    // Red ellipse with radial gradient color
    Ellipse ellipse = new Ellipse(100, // center X
        50 + 70/2, /* center Y */
        50,          /* radius X */
        70/2);       /* radius Y */
```

```
RadialGradient gradient1 = new RadialGradient(
    0,          /* focusAngle */
    .1,         /* focusDistance */
    80,         /* centerX */
    45,         /* centerY */
    120,        /* radius */
    false,       /* proportional */
    CycleMethod.NO_CYCLE, /* cycleMethod */
    new Stop(0, Color.RED), /* stops */
    new Stop(1, Color.BLACK)
);

ellipse.setFill(gradient1);
root.getChildren().add(ellipse);
double ellipseHeight = ellipse.getBoundsInParent()
    .getHeight();

// thick black line behind second shape
Line blackLine = new Line();
blackLine.setStartX(170);
blackLine.setStartY(30);
blackLine.setEndX(20);
blackLine.setEndY(140);
blackLine.setFill(Color.BLACK);
blackLine.setStrokeWidth(10.0f);
blackLine.setTranslateY(ellipseHeight + 10);

root.getChildren().add(blackLine);

// A rectangle filled with a linear gradient with a translucent color.
Rectangle rectangle = new Rectangle();
rectangle.setX(50);
rectangle.setY(50);
rectangle.setWidth(100);
rectangle.setHeight(70);
rectangle.setTranslateY(ellipseHeight + 10);

LinearGradient linearGrad = new LinearGradient(
    0, /* start X */
    0, /* start Y */
    0, /* end X */
    1, /* end Y */
    true, /* proportional */
    CycleMethod.NO_CYCLE, /* cycle colors stops */
    new Stop(0.1f, Color.rgb(255, 200, 0, .784)),
    new Stop(1.0f, Color.rgb(0, 0, 0, .784)));

```

```

rectangle.setFill(linearGrad);
root.getChildren().add(rectangle);

// A rectangle filled with a linear gradient with a reflective cycle.
Rectangle roundRect = new Rectangle();
roundRect.setX(50);
roundRect.setY(50);
roundRect.setWidth(100);
roundRect.setHeight(70);
roundRect.setArcWidth(20);
roundRect.setArcHeight(20);
roundRect.setTranslateY(ellipseHeight + 10 +
    rectangle.getHeight() + 10);

LinearGradient cycleGrad = new LinearGradient(
    50, /* start X */
    50, /* start Y */
    70, /* end X */
    70, /* end Y */
    false, /* proportional */
    CycleMethod.REFLECT, /* cycleMethod */
    new Stop(0f, Color.rgb(0, 255, 0, .784)),
    new Stop(1.0f, Color.rgb(0, 0, 0, .784))
);

roundRect.setFill(cycleGrad);
root.getChildren().add(roundRect);

primaryStage.setScene(scene);
primaryStage.show();
}

```

When specifying color values, the `PaintingColors.java` code uses the colors in the default RGB color space. To create a color, the code uses the `Color.rgb()` method. This method takes three integer values, representing red, green, and blue components. Another overloaded method takes three integer values and a fourth parameter with a data type of double. This fourth parameter is the alpha channel, which sets the opacity of the color. This value is between zero (0) and one (1). Zero is no opacity (transparent) and one is fully opaque. Keep in mind that there are other ways to create color such as HSB and Web. HSB stands for hue, saturation, and brightness. To create a color using HSB (color space), you would invoke the `Color.hsb()` method. Another way to specify color values that is common to web development in HTML and CSS is the use of RGB hexadecimal string values. For those developers who are familiar with this convention, they would use the `Color.web()` method.

After drawing the ellipse shape, the code invokes the `setFill()` method using a radial gradient to give the appearance of a 3D spherical object. Next, it creates a rectangle filled with a yellow semitransparent linear gradient. Added behind the yellow rectangle is a thick black line to demonstrate the semitransparent color. Finally, the code implements a rounded rectangle filled with a green-and-black reflective linear gradient resembling 3D tubes in a diagonal direction. Each shape and its associated color fill will be discussed in more detail in the following sections.

Gradient Color

Creating a gradient color in JavaFX involves five things:

1. A starting point to begin the first stop color (`javafx.scene.paint.Stop`).
2. The end point representing the end stop color.
3. The proportional property to specify whether to use standard screen coordinates or unit square coordinates.
4. The `Cycle` method to specify one of the three enums: `NO_CYCLE`, `REFLECT`, or `REPEAT`.
5. An array of stop (`Stop`) colors. Each stop containing a color will begin by painting the first stop color, then interpolating to the second stop color, and so on.

Stop Color

When working with gradient colors, you must have a list of at least two colors to interpolate over. Also, when dealing with gradients, the amount of gradient that can be specified between colors is called a *range offset*. To specify a range offset to distribute the colors, you need instances of the `Stop` class. The following code snippet shows seven stop colors having ranges of 0%, 15%, 30%, 45%, 60%, 75%, and 100%.

```
Stop rStop = new Stop(0.0, Color.RED);
Stop oStop = new Stop(0.15, Color.ORANGE);
Stop yStop = new Stop(0.30, Color.YELLOW);
Stop gStop = new Stop(0.45, Color.GREEN);
Stop bStop = new Stop(0.60, Color.BLUE);
Stop iStop = new Stop(0.75, Color.INDIGO);
Stop vStop = new Stop(1, Color.VIOLET);
```

The example code is a gradient that represents the colors of a rainbow (ROYGBIV), starting with the color red, which interpolates a range between 0% and 15%, then the color orange is a gradient between 15% to 30%. Next is a range from 30% to 45% using the color yellow and so on. Now that you know how to specify the range offset between stop colors, you will learn about the gradient axis, which helps linear gradients. The gradient axis is based on a line represented as start and end points, which assist in distributing the color across a shape.

Linear Gradient

When dealing with either linear or radial gradient color, pay special attention to the `proportional` attribute. By setting this attribute to `false`, you can draw a line (the gradient axis) having a beginning point (`start X`, `start Y`) and an end point (`end X`, `end Y`) based on standard screen (`x, y`) coordinates. An example is a linear gradient using a starting point of (0,0) and an ending point of (100,0), which creates a horizontal gradient (from left to right). If a rectangle is 100 pixels wide and has a stop color of black with a 0.0 offset and a stop color of white with a 1.0 offset, there would be an even distribution of the amount of color from black to white using screen coordinates between 0 and 100 pixels. But if for some reason you wanted to change the width of the rectangle, the gradient wouldn't scale properly. The gradient would not stretch the width of rectangle.

To allow the gradient to stretch based on the size of the shape, the `proportional` attribute is set to a value of `true` and the gradient axis line beginning and ending points will be represented as unit square coordinates. This means that x, y coordinates for the beginning and end points must be between 0.0 and 1.0 (`double`). This strategy is more compact and easier to define than screen coordinates on the scene graph. So, for instance, if a rectangle is 200 pixels wide and has the same two stop colors as mentioned, you would specify the gradient axis line with (0,0) and (1,0) for the gradient to stretch from left to right (the width of the shape).

Radial Gradient

The amazing thing about colors with gradients is that they can often make shapes appear three-dimensional. Gradient paint allows you to interpolate between two or more colors, which gives depth to the shape. JavaFX provides two types of gradients: a radial (`RadialGradient`) and a linear (`LinearGradient`) gradient. For this ellipse shape, you will be using a radial gradient (`RadialGradient`). This will give the ellipse the appearance of a spherical looking object.

Table 3-2 presents the JavaFX 8 Javadoc definitions found for the `RadialGradient` class. You can find this documentation at the following URL:

<http://download.java.net/jdk8/jfxdocs/javafx/scene/paint/RadialGradient.html>

Table 3-2. *RadialGradient Properties*

Property	Data Type	Description
<code>focusAngle</code>	<code>Double</code>	Angle in degrees from the center of the gradient to the focus point to which the first color is mapped.
<code>focusDistance</code>	<code>Double</code>	Distance from the center of the gradient to the focus point to which the first color is mapped.
<code>centerX</code>	<code>Double</code>	X coordinate of the center point of the gradient's circle.
<code>centerY</code>	<code>Double</code>	Y coordinate of the center point of the gradient's circle.
<code>radius</code>	<code>Double</code>	Radius of the circle defining the extent of the color gradient.
<code>proportional</code>	<code>boolean</code>	Coordinates and sizes are proportional to the shape this gradient fills.
<code>cycleMethod</code>	<code>CycleMethod</code>	Cycle method applied to the gradient.
<code>Stops</code>	<code>List<Stop></code>	Gradient's color specification.

In this example, the focus angle is set to zero, the distance is set to .1, the center X and Y are set to (80,45), the radius is set to 120 pixels, `proportional` is set to `false`, the cycle method is set to the no cycle (`CycleMethod.NO_CYCLE`), and the two color stop values are set to red (`Color.RED`) and black (`Color.BLACK`). These settings give a radial gradient to the ellipse by starting with the color red with a center position of (80, 45) (upper-left of the ellipse) that interpolates to the color black with a distance of 120 pixels (`radius`). As the radial gradient's center position is set to the color red, the gradient will interpolate to the color black, which appears as if the center position is an overhead light source with black mimicking a shadow (opposite of the light source).

Semitransparent Gradients

Next, you will see how to create the rectangle, which has a yellow semitransparent linear gradient. For the yellow rectangle you will use a linear gradient (`LinearGradient`) paint.

Table 3-3 presents the JavaFX 8 Javadoc definitions found for the `LinearGradient` class. You can find the definitions also at the following URL:

<http://download.java.net/jdk8/jfxdocs/index.html?javafx/scene/paint/LinearGradient.html>

Table 3-3. *LinearGradient Properties*

Property	Data Type	Description
<code>startX</code>	<code>Double</code>	X coordinate of the gradient axis start point.
<code>startY</code>	<code>Double</code>	Y coordinate of the gradient axis start point.
<code>endX</code>	<code>Double</code>	X coordinate of the gradient axis end point.
<code>endY</code>	<code>Double</code>	Y coordinate of the gradient axis end point.
<code>proportional</code>	<code>Boolean</code>	Whether the coordinates are proportional to the shape that this gradient fills. When set to true, use unit square coordinates; otherwise, use scene/screen coordinate system.
<code>cycleMethod</code>	<code>CycleMethod</code>	Cycle method applied to the gradient.
<code>stops</code>	<code>List<Stop></code>	Gradient's color specification.

To create a linear gradient paint, you specify `startX`, `startY`, `endX`, and `endY` for the start and end points. The start and end point coordinates denote where the gradient pattern begins and stops.

To create the second shape in Figure 3-15, the yellow rectangle, you set the start X and Y to (0.0, 0.0), end X and Y to (0.0, 1.0), proportional to true, the cycle method to no cycle (`CycleMethod.NO_CYCLE`), and two color stop values to yellow (`Color.YELLOW`) and black (`Color.BLACK`), with an alpha transparency of .784. These settings give a linear gradient to the rectangle from top to bottom with a starting point of (0.0, 0.0) (the top-left of a unit square) that interpolates to the color black to an end point of (0.0, 1.0) (the bottom-left of a unit square).

Reflective Cycle Gradients

Finally, at the bottom of Figure 3-15, you'll notice a rounded rectangle with a repeating pattern of a gradient using green and black in a diagonal direction. This is a simple linear gradient paint that is the same as the linear gradient paint (`LinearGradient`), except that the start X, Y and end X, Y values are set in a diagonal position, and the cycle method is set to reflect (`CycleMethod.REFLECT`). When you specify the cycle method to reflect (`CycleMethod.REFLECT`), the gradient pattern will repeat or cycle between the stop colors. The following code snippet implements the rounded rectangle having a cycle method of reflect (`CycleMethod.REFLECT`):

```
LinearGradient cycleGrad = new LinearGradient(
    50, // start X
    50, // start Y
    70, // end X
    70, // end Y
    false, // proportional
    CycleMethod.REFLECT, // cycleMethod
    new Stop(0f, Color.rgb(0, 255, 0, .784)),
    new Stop(1.0f, Color.rgb(0, 0, 0, .784))
);
```

Drawing Text

Another basic JavaFX node is the `Text` node, which enables you to display a string of characters on the scene graph. To create `Text` nodes on the JavaFX scene graph, you use the `javafx.scene.text.Text` class. Because all JavaFX scene nodes extend from `javafx.scene.Node`, they will inherit many capabilities such as the ability to be scaled, translated, or rotated.

Based on the Java inheritance hierarchy, the `Text` node's direct parent is a `javafx.scene.shape.Shape` class that provides even more capabilities than the `Node` class. Because a `Text` node is both a `Node` and a `Shape` object, it can perform geometric operations between two shape areas such as subtract, intersect, or union. You can also clip viewport areas with a shape similar to stenciled letters.

To demonstrate drawing text, in this section you will look at a basic example of how to draw `Text` nodes on the scene graph. This example touches on the following three capabilities:

- Positioning `Text` nodes using (x, y) coordinates
- Setting a `Text` node's stroke color
- Rotating a `Text` node about its pivot point

To make things a little interesting, you will create 100 `Text` nodes and produce random values for the three capabilities just mentioned. Figure 3-16 shows the drawing text example in action.

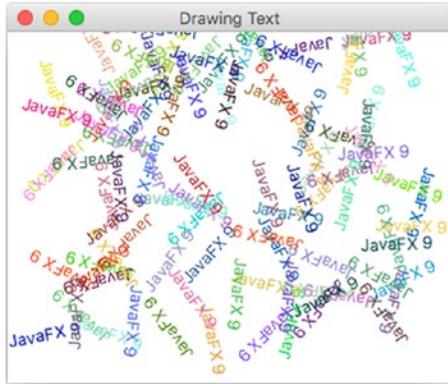


Figure 3-16. Drawing text

The `DrawingText.java` code shown in Listing 3-4 creates 100 `Text` nodes with randomly generated values for the following attributes:

- x, y coordinates
- RGB fill color
- Angle of rotation (degrees)

The code first creates a loop to generate random (x, y) coordinates to position `Text` nodes. In the loop, it creates random RGB color components between 0 and 255 to be applied to the `Text` nodes. Setting all components to 0 produces black. Setting all three RGB values to 255 produces the color white.

Listing 3-4. DrawingText.java

```

@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("Drawing Text");
    Group root = new Group();
    Scene scene = new Scene(root, 300, 250, Color.WHITE);
    Random rand = new Random(System.currentTimeMillis());
    for (int i = 0; i < 100; i++) {
        int x = rand.nextInt((int) scene.getWidth());
        int y = rand.nextInt((int) scene.getHeight());
        int red = rand.nextInt(255);
        int green = rand.nextInt(255);
        int blue = rand.nextInt(255);

        Text text = new Text(x, y, "JavaFX 9");

        int rot = rand.nextInt(360);
        text.setFill(Color.rgb(red, green, blue, .99));
        text.setRotate(rot);
        root.getChildren().add(text);
    }

    primaryStage.setScene(scene);
    primaryStage.show();
}

```

The rotation angle (in degrees) is a randomly generated value from 0–360 degrees, which causes the baseline of the text to be tilted. According to the API documentation, the `setRotate()` method will rotate about the pivot point, which is the center of the untransformed `layout bounds` (`layoutBounds`) property. Basically, the pivot point is the center of a node that has no transforms (scaling, translating, shearing, rotating, and so on) applied.

Note If you need to use a combination of transforms, such as `rotate`, `scale`, and `translate`, take a look at the `getTransforms().add(...)` method. For more details on the difference between bounds in local, bounds in parent, and layout bounds, see the Javadoc documentation. Also, check out Chapter 5 on layouts and the Scene Builder.

The following code is used in `DrawingText.java` to create random values for a `Text` node's x and y position (baseline), color, and rotation:

```

int x = rand.nextInt((int) scene.getWidth());
int y = rand.nextInt((int) scene.getHeight());
int red = rand.nextInt(255);
int green = rand.nextInt(255);
int blue = rand.nextInt(255);
int rot = rand.nextInt(360);

```

Once the random values are generated, they are applied to the Text nodes, which will be drawn onto the scene graph. Each Text node maintains a text origin property that contains the starting point of its baseline. In Latin-based alphabets, the baseline is an imaginary line underneath letters, similar to books on a bookshelf. However, some letters, such as the lowercase *j*, extend below the baseline. When specifying the *x* and *y* coordinate of the Text node you will be positioning the start of the baseline. In Figure 3-17, the *x* and *y* coordinates are located left of the lowercase *j* on the baseline in the text node javafx 9.



Figure 3-17. Text node's baseline (*x*, *y*) coordinate position

The following code snippet applies (*x*, *y*) coordinates, color (RGB) with an opacity .99, and a rotation (angle in degrees) to the Text node:

```
Text text = new Text(x, y, "JavaFX 9");
text.setFill(Color.rgb(red, green, blue, .99));
text.setRotate(rot);
root.getChildren().add(text);
```

You should be starting to see the power of the scene graph API in its ease of use, especially when working with Text nodes. Here, you were able to position, colorize, and rotate text. To spruce things up a bit, you will next see how to change a text's font and apply effects.

Changing Text Fonts

JavaFX's Font API enables you to change font styles and font sizes in the same way as word processing applications. To demonstrate this, I created a JavaFX application that displays four text nodes with the string value of "JavaFX 9 by Example", each having a different font style. In addition to the font styles, I also added effects such as a drop shadow (DropShadow) and reflection (Reflection).

Figure 3-18 shows the example's output, and Listing 3-5 shows the *ChangingTextFonts.java* source code.

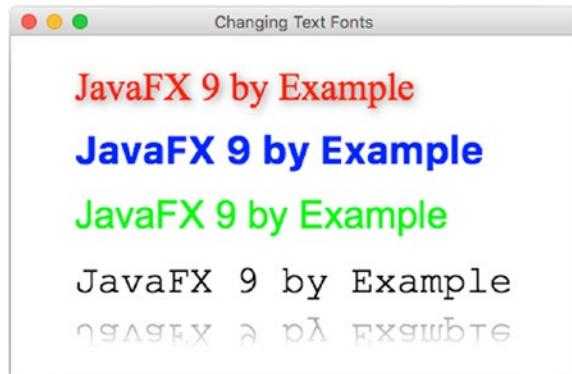


Figure 3-18. Changing text fonts

Listing 3-5. ChangingTextFonts.java

```
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("Changing Text Fonts");

    System.out.println("Font families: ");
    Font.getFamilies()
        .stream()
        .forEach( i -> System.out.println(i));
    System.out.println("Font names: ");
    Font.getFontNames()
        .stream()
        .forEach( i -> System.out.println(i));

    Group root = new Group();
    Scene scene = new Scene(root, 580, 250, Color.WHITE);

    // Serif with drop shadow
    Text text2 = new Text(50, 50, "JavaFX 9 by Example");
    Font serif = Font.font("Serif", 30);
    text2.setFont(serif);
    text2.setFill(Color.RED);
    DropShadow dropShadow = new DropShadow();
    dropShadow.setOffsetX(2.0f);
    dropShadow.setOffsetY(2.0f);
    dropShadow.setColor(Color.rgb(50, 50, 50, .588));
    text2.setEffect(dropShadow);
    root.getChildren().add(text2);

    // SanSerif
    Text text3 = new Text(50, 100, "JavaFX 9 by Example");
    Font sanSerif = Font.font("SanSerif", 30);
    text3.setFont(sanSerif);
    text3.setFill(Color.BLUE);
    root.getChildren().add(text3);

    // Dialog
    Text text4 = new Text(50, 150, "JavaFX 9 by Example");
    Font dialogFont = Font.font("Dialog", 30);
    text4.setFont(dialogFont);
    text4.setFill(Color.rgb(0, 255, 0));
    root.getChildren().add(text4);

    // Monospaced
    Text text5 = new Text(50, 200, "JavaFX 9 by Example");
    Font monoFont = Font.font("Monospaced", 30);
    text5.setFont(monoFont);
    text5.setFill(Color.BLACK);
    root.getChildren().add(text5);
```

```

// Reflection
Reflection refl = new Reflection();
refl.setFraction(0.8f);
refl.setTopOffset(5);
text5.setEffect(refl);

primaryStage.setScene(scene);
primaryStage.show();
}

```

Note Listing 3-5, `ChangingTextFonts.java`, presents only the `start()` method, which is the heart of the example application. To see the full code listing, download the book's full example code from [Apress.com](#).

With Text nodes, JavaFX takes a retained-mode approach, in which nodes are using vector-based graphics rendering instead of an immediate mode rendering. The immediate mode uses bitmap graphic rendering strategies. By using vector-based graphics you will have nifty advantages over bitmap graphics. The major advantage is that they allow you to scale shapes and apply different effects without pixilation (the jaggies). For example, in an immediate mode rendering the image becomes grainy when scaled larger. However, in retained mode, you will have smooth (anti-aliased) shapes. It's nice to be able to see beautiful type fonts (typography) that are smooth at all different sizes.

`ChangingTextFonts.java` focuses on the following JavaFX classes to be applied to the Text node. The code uses the Serif, SanSerif, Dialog, and Monospaced fonts along with the drop shadow and reflection effects.

- `javafx.scene.text.Font`
- `javafx.scene.effect.DropShadow`
- `javafx.scene.effect.Reflection`

The code begins by setting the title on the stage. Next, you will notice the new Java 8 language lambdas feature at work, where the code lists the current system's available font families and font names. If you are new to lambdas, in Chapter 4 you will take a closer look at the concept, but for right now you can think of it as an elegant way to iterate over collections. The font family and font name list will be printed on the console output. This is very convenient for you to later experiment with different font styles. The following lines list the available fonts on the current system using Java 8's lambda syntax:

```

System.out.println("Font families: ");
Font.getFamilies()
    .stream()
    .forEach( i -> System.out.println(i));

System.out.println("Font names: ");
Font.getFontNames()
    .stream()
    .forEach( i -> System.out.println(i));

```

Note In case the selected font is not installed on your system, Text nodes will use a default system font. There are many sites that offer fonts that you can download and install. Check out Google Fonts at <https://www.google.com/fonts>.

After listing the available fonts, the code creates a root node with a background color for its scene. Before drawing the first Text node, let's quickly discuss how to obtain a font. To obtain a font, the code invokes the static method `font()` from the `Font` class. When calling the `font()` method, you can specify the font name and the point size to return a suitable font to the caller. A font name is a string value that represents a system's supported font types. Refer to the Javadoc documentation to see the other ways to obtain system fonts. The following lines show the creation of a Text node instance and obtaining a 30 point Serif font. Once the font is obtained, the Text node `setFont()` method is used to apply the font.

```
Text text = new Text(50, 50, "JavaFX 9 by Example");
Font serif = Font.font("Serif", 30);
text.setFont(serif);
```

Note Although Listing 2-5 uses absolute positioning of text nodes, at some point you may want to display text nodes with the ability to wrap several text nodes while keeping their own font formatting in the same layout. To achieve this behavior, refer to JavaFX 8's new `TextFlow` API. Refer to the Javadoc documentation at: <https://docs.oracle.com/javase/8/javafx/api/javafx/scene/text/TextFlow.html>

Applying Text Effects

In the first Text node in the `ChangingTextFonts.java` example, the code adds a drop shadow effect. A common graphics trick for adding a drop shadow is to create at least two layers of the image, with the bottom image colored dark with a slight offset. The dark image, slightly offset, will mimic a shadow. In JavaFX, a drop shadow is an actual effect (`DropShadow`) object and is applied to a single Text node instance without the need of multiple images layered. The `DropShadow` object is set to be positioned based on an x, y offset in relation to the Text node. You can also set the color of the shadow; here, the code will set the shadow color to gray with 0.588 opacity. Opacity is a range between 0 and 1 (double), where 0 is transparent and 1 is fully opaque. The following is an example of setting a Text node's effect property with a drop shadow effect (`DropShadow`):

```
DropShadow dropShadow = new DropShadow();
dropShadow.setOffsetX(2.0f);
dropShadow.setOffsetY(2.0f);
dropShadow.setColor(Color.rgb(50, 50, 50, .588));
text2.setEffect(dropShadow);
```

Although this example is about setting text fonts, it also demonstrates applying a drop shadow effect to Text nodes. The `ChangingTextFonts.java` example includes yet another effect (just kicking it up a notch). While creating the last Text node using the monospaced font, I've applied the popular reflection effect. Calling the `setFraction()` method with 0.8f is essentially specifying that you want 80 percent of the reflection to be shown. The reflection values range from zero (0%) to one (100%). In addition to the fraction to be shown, the reflection's gap or top offset can be set. In other words, the space between the opaque node portion and the reflection portion is adjusted by the `setTopOffset()` method. The top offset defaults to zero. The following code snippet implements a reflection of 80% with a float value of 0.8f and a top offset of five pixels:

```
Reflection refl = new Reflection();
refl.setFraction(0.8f);
refl.setTopOffset(5);
text5.setEffect(refl);
```

Summary

In this chapter, you learned the fundamentals of drawing 2D shapes on the JavaFX scene graph. You gained insight into the differences between the Cartesian and screen coordinate systems. This furthered your knowledge of how to draw shape nodes onto the JavaFX scene graph. The first example was the basic shape of a line using the JavaFX `Line` class. Here, you delved deeper into the common properties of the parent class (`Shape`). Some of the common properties discussed were setting the shape's stroke width, stroke color, and dashed pattern. Next, you explored simple, complex, and custom shapes by drawing fun cartoon-like images. After creating shapes, you learned how to paint them using colors. You not only used standard RGB colors, you also used various built-in techniques such as a linear (`LinearGradient`) and radial (`RadialGradient`) gradient paint. Finally, you worked with JavaFX's `Text` node. Using the `Text` node, you were able to obtain the available system fonts and apply effects such as drop shadow (`DropShadow`) and reflection (`Reflection`).

In the last example, which detailed changing text fonts, you learned how to iterate over a list of system fonts using Java 8's new stream API and lambda expressions. In Chapter 4, you will be introduced to Java 8's lambdas, streams, and properties.

CHAPTER 4



Lambdas and Properties

In this chapter, you learn about the new language feature introduced in Java 8 called *lambda expressions*, or *lambdas*. The chapter also covers JavaFX properties and binding APIs. The goal of this chapter is to demonstrate how lambdas and properties are used in the context of JavaFX GUI applications. Having said this, I mainly concentrate on the common features that are used in most of the examples in this book and do not detail every lambda and properties feature.

To get a better idea of Java's lambda roadmap, visit the following article by Brian Goetz, who is Oracle's Java Language Architect:

<http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html>

The term *lambda* derives from what is known in mathematics as *lambda calculus*. The general concepts from that field are used to derive the specific behaviors that have been added to Java, which are described in this chapter. Figure 4-1 shows the Greek letter lambda, which you sometimes see in connection with the topic.



Figure 4-1. The Greek symbol for lambda

Lambda

Java Lambda is based on JSR (Java Specification Request) 335, “Lambda Expressions for the Java™ Programming Language.” The feature was appropriately named after lambda calculus, the formal system in mathematical logic and computer science to express computations. JSR 335, better known as *Project Lambda*, comprised many features, such as expressing parallel calculations on streams (the Stream API).

A primary goal of lambdas is to help address the lack in the Java language of a good way to express functional programming concepts. Languages that support functional programming concepts have the ability to create anonymous (unnamed) functions, similar to creating objects instead of methods in Java. These function objects are commonly known as *closures*. Some common languages that support closures or lambdas are Common Lisp, Clojure, Erlang, Haskell, Scheme, Scala, Groovy, Python, Ruby, and JavaScript.

The main idea is that languages that support functional programming will use a closure-like syntax. In Java 8, you can create anonymous functions as first-class citizens. In other words, functions or closures can be treated like objects, so that they can be assigned to variables and passed into other functions. In order to mimic closures in the previous Java world (Java 1.1 to Java 7), you would typically use anonymous inner classes to encapsulate functions (behavior) as objects. Listing 4-1 shows an example of an anonymous inner class that defines handler code for a JavaFX button when pressed (prior to Java 8).

Listing 4-1. A Button's OnAction Set with an Anonymous Inner Class

```
Button btn = new Button();
btn.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        System.out.println("Hello World");
    }
});
```

You will notice that this code looks very verbose just to wire up a button. Buried deep in an anonymous inner class is a single line to output text. Wouldn't it be nice to be able to express a block of code containing the behavior you want without the need of so much boilerplate code? Look no further. Java 8 solves this issue with the use of lambda expressions (closures). To see how it looks with Java 8 lambda expressions, let's rewrite the button handler code. Listing 4-2 is the button's EventHandler code rewritten as a Java 8 lambda expression.

Listing 4-2. A Button's OnAction Set with a Lambda Expression

```
btn.setOnAction(event -> System.out.println("Hello World") );
```

Using lambda expressions not only makes code concise and easy to read, but the code is also likely to perform better. Actually, under the hood, the compiler is capable of optimizing code and likely to be able to reduce its footprint.

Lambda Expressions

As discussed, lambda expressions are Java 8's version of closures. In the same way other languages create closures syntactically, you will also be able to reference them and pass them into other methods. In order to achieve this ability, you will later learn about *functional interfaces*. But for now, let's look at the basic syntax for lambda expressions.

Syntax

There are two ways to specify lambda expressions. The following simple examples illustrate the general forms:

```
(param1, param2, ...) -> expression;
(param1, param2, ...) -> { /* code statements */ };
```

Lambda expressions begin with a list of parameters surrounded by parentheses, followed by the arrow symbol `->` (a hyphen and a greater-than sign) and an expression body. Similar to Java methods, the parentheses are for parameters that are passed into an expression. The surrounding parentheses are *optional* only if there is one parameter defined. When the expression doesn't accept parameters (and is said to be *empty*), the parentheses are still required.

Separating the parameter list and the expression body is the *arrow symbol*. The expression body or code block may or may not be surrounded by curly braces. When the expression body doesn't have surrounding curly braces, it must consist of only one statement. When an expression is a one-line statement, it is evaluated and returned to the caller implicitly. Just remember that if the method requires a return type and your code block has curly braces, you must have a `return` statement. Listing 4-3 shows two equivalent lambda expressions, with and without curly braces.

Listing 4-3. One Statement with an Explicit Return, and the Other with an Implicit Return

```
// explicit return of result
Function<Double, Double> func = x -> { return x * x; }

// evaluates & implicitly returns result
Function<Double, Double> func = x -> x * x;

double y = func(2.0); // x = 4.0
```

Another thing to note is that parameters can be *optionally typed*. The compiler will infer the type of the parameters depending on the context. In Listing 4-3, a functional interface of type `java.util.function.Function<T, U>` where a `T` type is transformed into a type `U`. There you will notice the parameter `x` doesn't have a type `Double` specified. That's because the Java 8 compiler will infer the type based on the functional interface definition.

Let's look at an example of passing a lambda expression to a method. A simple example would be setting a JavaFX button by using the `setOnAction()` method. As mentioned, the compiler can figure out the lambda parameters by matching the method's argument types (its *signature*) and its return type. Sometimes for readability, you can optionally specify the type of each parameter. If you specify the type, you will be required to surround the list of parameters in parentheses. The following line specifies the type (`ActionEvent`) of the parameter for the lambda expression for a JavaFX button's `setOnAction()` method.

```
btn.setOnAction( ActionEvent event) -> System.out.println("Hello World") );
```

You will also notice that it is okay to leave off the semicolon for a one line statement that is inside methods, such as the `setOnAction()` method. One last piece of syntactic sugar to note is that if you have one parameter to be passed into your lambda, the parentheses can be omitted. The following is a lambda expression that has one parameter (`event`) without parentheses.

```
btn.setOnAction( event -> System.out.println("Hello World") );
```

As a recap of the syntax variations just discussed, the statements in Listing 4-4 are all equivalent lambda expressions.

Listing 4-4. Three Syntactically Equivalent Lambda Expressions to Set Action Code on a JavaFX Button

```
btn.setOnAction( ActionEvent event) -> {System.out.println(event); } );
btn.setOnAction( event) -> System.out.println(event) );
btn.setOnAction( event -> System.out.println(event) );
```

Method Reference

Listing 4-4 shows three ways to express a lambda expression for an `ActionEvent` for a JavaFX button, but did you know there is a fourth way?

Often when using lambdas, there are methods that take a single parameter as input or a single value as a return type. This is very redundant so, new in Java 8 is the concept of *method references*. Method references are basically syntactic sugar that allow you to make method calls with even less verbosity, subsequently making things easier to read. For example, the following is a lambda expression that uses a method reference:

```
btn.setOnAction(System.out::println);
```

This code sets the action on the button, and you'll notice it's a concise version that behaves the same way as Listing 4-4. The difference is that there isn't a input parameter for the `ActionEvent` for the lambda and an unusual double colon between the `System.out` and the `println` method. The double colon is called the *scope operator*, and it references the method by name. You'll also notice the `println` method's parentheses are absent. If you remember, whenever a lambda expression takes a single parameter as input, the parameter is implicitly passed to the method `println` that takes a single input. Of course, both types must be the same. The event object in the example will implicitly call `toString()` to pass a `String` to the `println` method.

There are other method reference types beyond the scope of this book. See the Javadoc documentation for other use-case scenarios. Check out the following tutorials on method references:

<https://docs.oracle.com/javase/tutorial/java/java00/methodreferences.html>

Variable Capture

Typically, in functional programming languages, the use of closures (lambda expressions) allows you to use variables outside the closure's scope. These variables are often described as *nonlocal* or *capture* variables because they aren't locally scoped to the lambda expression or method. When these nonlocal variables are used in lambda expressions, they need to be immutable. This is a key principle in the functional programming world. New to the Java 8 compiler is the ability to reference variables outside the scope of an anonymous inner class or lambda expression.

When using lambda expressions, you can optionally reference variables scoped outside the closure function. Typically, prior to Java 8, these nonlocal variables needed to be `final` (immutable). This immutable state enables the compiler to better predict program behavior and thus be able to optimize code. In Java Swing development, this occurred often when variables of the enclosing scope needed to be used inside the scope of an anonymous inner class method (local variable scope). In order to abide by the compiler rules, you had to declare a variable as `final`.

The Java 8 compiler has a new capability that will infer the intent of nonlocal variables passed into anonymous inner class methods and lambda expressions to then be converted to `final` (immutable). To see how the old-school Java Swing developers (prior to Java 8) used to receive compile-time errors, look at Listing 4-5. The scenario is a button that will modify the text of a Swing `JLabel` component after the button is pressed. Prior to Java 8, the code would not compile correctly, because the `label` variable needed to be declared `final`.

Listing 4-5. Code That Would Receive a Compiler Error Prior to JDK 8

```
// Prior to Java 8
// label variable is a nonlocal variable to actionPerformed()
JLabel label = new JLabel("Press Me");
btn.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        label.setText("Pressed"); /* !Compile Error! Variable label must be final. */
    }
});
```

New to Java 8 is the ability to convert nonlocal variables to be final. In Listing 4-6, you'll notice that the code is identical except for the comments. Starting with Java 8, you can now use the variable capture capability with anonymous inner classes and lambda expressions.

Listing 4-6. Code That Compiles Successfully in JDK 8

```
// New in Java 8
// label variable is a nonlocal variable to actionPerformed()
JLabel label = new JLabel("Press Me");
// compiler implicitly converts label as a final (immutable) variable
btn.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        label.setText("Pressed"); /* Legal. Variable label is converted to be final */
    }
});
```

Both of the code examples shown in Listing 4-6 are identical. The difference comes when using a JDK 8 or later compiler. When using the Java 8 compiler, the capture variable will implicitly become final (immutable reference), and that will occur without triggering any compilation error.

Functional Interfaces

The advent of cloud computing has helped to reinvigorate many functional programming languages. It became apparent there was a paradigm shift in problem-solving that involves extremely large datasets. A typical use case when applying functional programming techniques is the ability to iterate over datasets while performing computations in a distributed fashion so that load can be shared among nodes or CPU cores. In contrast, imperative programming languages gather data to then be passed into a tight `for` loop to be processed. Because of how data and code are coupled, this puts a lot of the burden on one thread (core) to process so much data. The problem needs to be decomposed to allow other threads (cores) to participate in the computation, which then becomes distributed.

One of the advantages of functional programming is the ability to express functionality in a syntactically concise manner, but more important is the ability to pass functionality (lambda expressions) to methods. Being able to pass lambda expressions to methods often fosters the concept of *lazy evaluation*. This behavior is the same as function callback behavior (asynchronous message passing), where invocations are deferred (and thus “lazy”) until a later time. The opposite of lazy evaluation is *eager evaluation*. Using lazy evaluations will often increase performance by avoiding unnecessary calculations.

Another important topic to discuss is how functional interfaces are used as closures in the Java language. Instead of implementing functions as first-class types in Java, the designers/architects of the Java Language cleverly defined the notion of functional interfaces as closures. Simply put, a functional interface is basically a single abstract method (SAM). The idea of functional interfaces has been around for a very long time. For instance, those who have worked with Java threads will recall using the `Runnable` interface,

where there is a single `run()` method with a `void` return type. The single abstract method pattern is an integral part of Java 8's lambda expressions. As an example of a functional interface, I created an interface called `MyEquation` with a single abstract `compute()` method. Once it is created, you can declare and assign variables with a lambda expression. Listing 4-7 is a functional interface that has a single abstract `compute()` method.

Listing 4-7. A Declared Functional Interface with a Single Abstract Method, `compute()`

```
// functional interface
interface MyEquation {
    double compute(double val1, double val2);
}
```

After creating a functional interface, you can declare a variable to be assigned with a lambda expression. Listing 4-8 demonstrates the assignment of lambda expressions to functional interface variables.

Listing 4-8. Two Variables of a Functional Interface, Assigned with Lambda Expressions

```
MyEquation area = (height, width) -> height * width;
MyEquation perimeter = (height, width) -> 2*height + 2*width;

System.out.println("Area = " + area.compute(3, 4));
System.out.println("Perimeter = " + perimeter.compute(3, 4));
```

The following is the output of Listing 4-8:

```
Area = 12.0
Perimeter = 14.0
```

Aggregate Operations

Another powerful use of Java 8's lambda expressions is working with collections. Java 8 introduces the new Stream API (`java.util.stream.*`), which allows you to process elements from a given source. A source can be a reference to data structures such as collections or IO channels. According to Oracle's Java 8 documentation (Javadoc) on the Stream API, “a stream is not a data structure that stores elements; instead, they carry values from a source (which could be a data structure, a generator, an IO channel, etc.) through a pipeline of computational operations.”

A pipeline is a sequence of operations (lambda expressions/functional interfaces) that can process or interrogate each element in a stream. Such operations allow you to perform aggregate tasks. Aggregate operations are similar to the way spreadsheets can execute some computation over a series of cells, such as formatting, averaging, or summing up values. To begin using aggregate operations on collections, you will first invoke the `default stream()` method on the `java.util.Collection` interface. You'll learn more about default methods shortly, but for now all collections (`Collection`) have a method `stream()` that returns a `java.util.stream.Stream` instance. Listing 4-9 is a snippet of code that obtains a stream of elements from a source collection (`List<Integer>`).

Listing 4-9. A Stream Obtained by Invoking the `stream()` Method from a Collection

```
List<Integer> values = Arrays.asList(23, 84, 74, 85, 54, 60);
Stream<Integer> stream = values.stream();
```

The common built-in aggregate operations are `filter`, `map`, and `forEach`. A `filter` allows you to pass in an expression to filter elements and returns a new `Stream` containing the selected items. The `map` operation converts (or maps) each element to another type and returns a new `Stream` containing items of the mapped type. For instance, you may want to map `Integer` values to `String` values of a stream. A `forEach` operation allows you to pass in a lambda expression to process each element in the stream.

A typical use case that exercises these three common aggregate operations is this: given a collection of integers, filter on values greater than a threshold value (a nonlocal variable), convert filtered items to hex values, and print each hex value. Listing 4-10 implements the use case, which exercises various aggregate operations.

Listing 4-10. The Use of Aggregate Operations Over a Collection of Integers

```
// create a list of values
List<Integer> values = Arrays.asList(23, 84, 74, 85, 54, 60);
System.out.println("values: " + values.toString());

// nonlocal variable to be used in lambda expression.
int threshold = 54;
System.out.println("Values greater than " + threshold + " converted to hex:");

Stream<Integer> stream = values.stream();

// using aggregate functions filter() and forEach()
stream
    .filter(val -> val > threshold) /* Predicate functional interface */
    .sorted()
    .map(dec -> Integer.toHexString(dec).toUpperCase() ) /* Consumer functional interface*/
    .forEach(val -> System.out.println(val)); /* each output values. */
```

Following is the output from Listing 4-10:

```
values: [23, 84, 74, 85, 54, 60]
Values greater than 54 converted to hex:
3C
4A
54
55
```

Note This output is a sorted list of integers ascending displaying the hexadecimal value. The decimal value equivalence are as follows: 3C=60, 4A=74, 54=84, and 55=85.

In Listing 4-10, the code begins by creating a list of integers as a collection. Next, it outputs the original list of elements (unmodified). Then the code declares a nonlocal variable to the lambda expression called `threshold`, to be used in a filter expression. Next, you obtain a stream from the `values` collection (via `stream()` method) to perform aggregate operations.

The first operation is a filter that receives a lambda expression to filter elements (integers) greater than 54 (`val > threshold`). The next operation is the `sorted()` method, which sorts elements that were returned from the filter method stream. Continuing the method chaining, notice the `map()` operation. The `map()` method operation can map each element in the stream from one data type to another.

In this scenario, the stream elements are `Integer` objects, which are mapped to `String` objects. The `map()` method returns another stream, containing elements of type `Stream<String>`. In this example, the string elements are hexadecimal values. Finally, the `forEach()` operation iterates over each element in the stream of string (hex) elements to be printed. Keep in mind that the `forEach()` method is used for non-parallel processing of streams.

You'll notice that as each aggregate function is called, the return type is a `Stream` object, which allows you to method-chain operations in a way very similar to the *builder pattern*. To clarify, the idea of method chaining actually comes from the fluent interface pattern for API design.

Although I've only touched on the main features of the `Stream` API, I didn't mention the important topic of parallelism. *Parallelism* is the ability to distribute work to CPU cores concurrently, thus reducing the overall computational time. Modern desktops, smartphones, tablets, and computers typically have multi-cores. To take advantage of true parallel processing, the Java 8 platform now supports parallel streams. To perform operations in parallel, you can simply invoke the method `parallelStream()` method on a collections. Since the items can be manipulated concurrently, it is advised to use a thread-safe collection. In Listing 4-11, the list of integers will be wrapped using the `Collections.synchronizedList()` method.

Listing 4-11. Aggregate Operations Example Written to Support Parallel Streams

```
/**
 * Aggregate Operations.
 * @author cdea
 */
public class AggregateOperations {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // create a list of values
        List<Integer> values = Collections.synchronizedList(
            Arrays.asList(23, 84, 74, 85, 54, 60));
        System.out.println("values: " + values.toString());

        // non-local variable to be used in lambda expression.
        int threshold = 54;
        System.out.println("Values greater than " + threshold + " converted to hex:");
        java.util.stream.Stream<Integer> stream = values.parallelStream();

        // using aggregate functions filter() and forEach()
        stream
            .filter(val -> val > threshold) /* Predicate functional interface */
            .sorted()
            .map(dec -> Integer.toHexString(dec).toUpperCase() ) /* Consumer functional interface */
            .forEachOrdered(System.out::println);
    }
}
```

After making a collection synchronized, the code obtains the `Stream` reference as a parallel stream by invoking the `parallelStream()` method. After performing immediate or *aggregate functions* against the parallel stream, the last *terminal function* invoked is the `forEachOrdered()` method. You'll notice instead of the `forEach()` method the `forEachOrdered()` method will iterate over each item when parallel processing is complete (ordering). If you use the `forEach()` method, the items will be out of order.

Note When using parallel streams to be sorted, use the terminal function `forEachOrdered()` instead of `forEach()`. According to the Javadoc, it states “Performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.”

Did you know the parallel code example is likely to be slower than using non-parallel stream? That's because the example uses so few items in its collection. The parallel code will create new threads per CPU core to handle each computation. Having so few items to process the overhead of each thread's context switching and scheduling is likely to be slower than if you had a single thread looping through the elements.

Now, if you had tons of data, parallel streams will outperform the single thread using one core. As a rule of thumb, you'll want to use parallel streams whenever you have extremely large datasets, but when you have a very small amount of data use the non-parallel stream. Typically, if you have four cores and you have more than 100,000 items, use parallel streams.

Default Methods

Default methods are a new way to add default implementation methods to Java interfaces. You're probably scratching your head and thinking out loud, “Java interfaces can have implementations?!” Yes, Java 8 now has support for the concept, called *virtual extension methods*, better known as *defender methods*. Default methods have the effect of adding (extending) new behavior to interfaces without breaking compatibility. Default methods are not abstract methods but methods having implementation (code).

For example, adding new abstract methods to Java interfaces can affect all implementation classes. Because of the strict contract of Java interfaces, the compiler forces the implementer to implement abstract methods, but classes that implement interfaces with default methods do not force the developer to implement those default methods. Rather, derived classes will acquire the behavior of the default method implementation.

An Example Case: Cats Large and Small

It might not be obvious, but the method `stream()` on the `Collection` interface used earlier for aggregate operations isn't an abstract method, but actually a Java 8 default method. Listing 4-12 is the default method `stream()` in the `java.util.Collection` interface from the Java 8 source code.

Listing 4-12. The Default Method `stream()` of Java 8's `Collection` Interface

```
default Stream<E> stream() {
    return StreamSupport.stream(splitterator, false);
}
```

For this example, I created a domain model (consisting of classes and interfaces) about *cats* (grrr, meow). The cat kind or species can easily demonstrate default behavior that is found in some cat kinds and not others. For instance, the great cats such as lions, tigers, and jaguars are able to roar; whereas smaller cats can purr. Also, a house (domestic) cat is able to meow. To further describe these behavioral traits, look at Figure 4-2, a class diagram depicting default behavior between cat kinds.

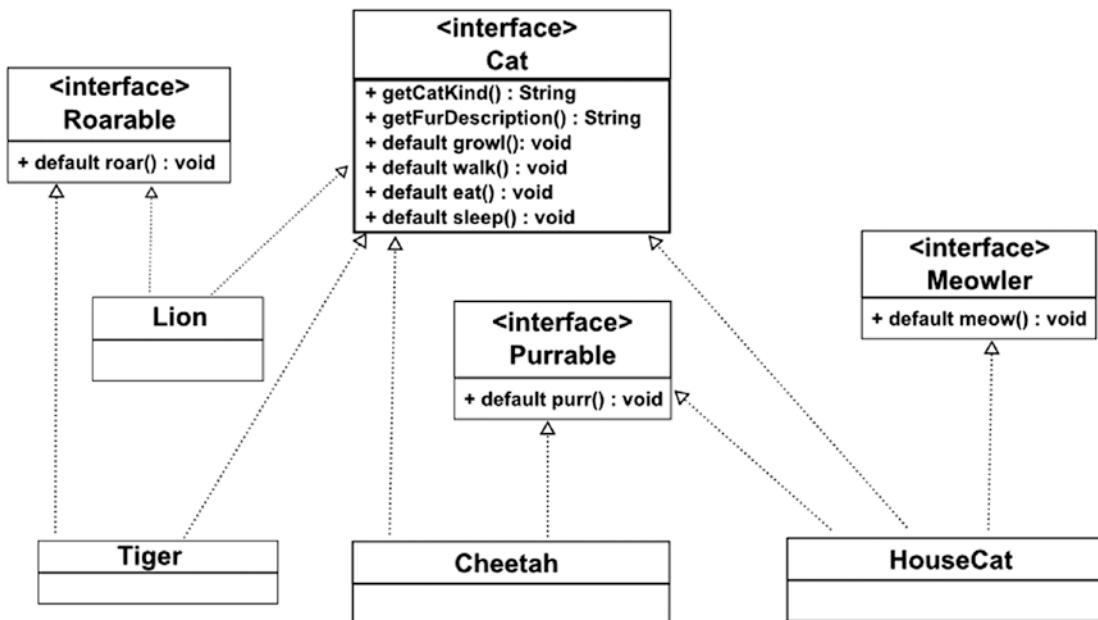


Figure 4-2. Cat domain model represented in a UML class diagram

Looking at the Tiger and Lion classes, you'll notice they each implement both Roarable and Cat interfaces. Both tigers and lions have the same behavior (can roar) via the default `roar()` method from the Roarable interface. Keep in mind that the Roarable interface doesn't have any abstract methods, but rather a default method `roar()`, which both Lion and Tiger acquire without implementing it.

In contrast, if the Roarable interface had `roar()` as an abstract method (a pure virtual function), you would have to implement `roar()` in both Tiger and Lion's concrete classes. With both having the same functionality in two places, down the road things could become a major maintenance nightmare. That's why default methods allow you to add behavior without having to modify derived classes.

Code for the Example

To see the class diagram in Figure 4-2 implemented as code, refer to Listings 4-13 through 4-20. To implement the class diagram, I briefly outline the three parts of the code example. First you will see the set of cat interfaces with default method behaviors. The concrete cat classes such as Lion, Cheetah, and HouseCat are shown second. Last is the main example application, which demonstrates default methods in action.

Listings 4-13 through 4-16 contain the UML interfaces Roarable, Purrable, Meowler, and Cat, each containing a default method.

Listing 4-13. The Roarable Interface with a Default Method roar()

```
public interface Roarable {
    default void roar() {
        System.out.println("Roar!!!");
    }
}
```

Listing 4-14. The Purrable Interface with a Default Method purr()

```
public interface Purrable {
    default void purr() {
        System.out.println("Purrrrrr...\"");
    }
}
```

Listing 4-15. The Meowler Interface with a Default Method meow()

```
public interface Meowler {
    default void meow() {
        System.out.println("MeeeeOww!");
    }
}
```

Listing 4-16. The Main Cat Interface with Abstract and Default Methods

```
/**
 * This represents an abstract Cat containing default methods common to all cats.
 * @author carldea
 */
public interface Cat {
    String getCatKind();
    String getFurDescription();
    default void growl() {
        System.out.println("Grrrrowl!!!");
    }
    default void walk() {
        System.out.println(getCatKind() + " is walking.");
    }
    default void eat() {
        System.out.println(getCatKind() + " is eating.");
    }
    default void sleep() {
        System.out.println(getCatKind() + " is sleeping.");
    }
}
```

Listings 4-17 through 4-20 show the concrete implementation classes `Tiger`, `Lion`, `Cheetah`, and `HouseCat`, which utilize the interfaces just shown.

Listing 4-17. The Tiger Class Implementing the Cat and Roarable Interfaces

```
public class Tiger implements Cat, Roarable {

    @Override
    public String getCatKind() {
        return getClass().getSimpleName();
    }

    @Override
    public String getFurDescription() {
        return "striped";
    }
}
```

Listing 4-18. The Lion Class Implementing the Cat and Roarable Interfaces

```
public class Lion implements Cat, Roarable {
    @Override
    public String getCatKind() {
        return getClass().getSimpleName();
    }

    @Override
    public String getFurDescription() {
        return "gold-brown";
    }
}
```

Listing 4-19. The Cheetah Class Implementing the Cat and Purrable Interfaces

```
public class Cheetah implements Cat, Purrable {
    @Override
    public String getCatKind() {
        return getClass().getSimpleName();
    }

    @Override
    public String getFurDescription() {
        return "spotted";
    }
}
```

Listing 4-20. The HouseCat Class Implementing the Cat, Purrable, and Meowler Interfaces

```
public class HouseCat implements Cat, Purrable, Meowler {
    @Override
    public String getCatKind() {
        return "Domestic Cat";
    }

    @Override
    public String getFurDescription() {
        return "mixed brown and white";
    }
}
```

Listing 4-21 is the main application, which invokes all of the methods of cat instances, demonstrating their behaviors.

Listing 4-21. The Main Application File (Mixins.java) That Executes the Code Example

```
/**Demonstrates default methods in Java 8.
 * Mixins.java
 *
 * @author cdea
 */
public class Mixins {
    public static void main(String[] args) {
        Tiger bigCat = new Tiger();
        Cheetah mediumCat = new Cheetah();
        HouseCat smallCat = new HouseCat();
        System.out.printf("%s with %s fur.\n", bigCat.getCatKind(),
            bigCat.getFurDescription());
        bigCat.eat();
        bigCat.sleep();
        bigCat.walk();
        bigCat.roar();
        bigCat.growl();
        System.out.println("-----");
        System.out.printf("%s with %s fur.\n", mediumCat.getCatKind(),
            mediumCat.getFurDescription());
        mediumCat.eat();
        mediumCat.sleep();
        mediumCat.walk();
        mediumCat.growl();
        mediumCat.purr();
        System.out.println("-----");
        System.out.printf("%s with %s fur.\n", smallCat.getCatKind(),
            smallCat.getFurDescription());
        smallCat.eat();
        smallCat.sleep();
        smallCat.walk();
        smallCat.growl();
        smallCat.purr();
        smallCat.meow();
        System.out.println("-----");
    }
}
```

Following is the output from Listing 4-20, which demonstrates default method invocations:

```
run:
Tiger with striped fur.
Tiger is eating.
Tiger is sleeping.
Tiger is walking.
Roar!!
```

```

Grrrrowl!!
-----
Cheetah with spotted fur.
Cheetah is eating.
Cheetah is sleeping.
Cheetah is walking.
Grrrrowl!!
Purrrrrrr...
-----
Domestic Cat with mixed brown and white fur.
Domestic Cat is eating.
Domestic Cat is sleeping.
Domestic Cat is walking.
Grrrrowl!!
Purrrrrrr...
MeeeeOww!
-----
BUILD SUCCESSFUL (total time: 0 seconds)

```

Explanation of the Code

Figure 4-2, shown earlier, depicts a UML class diagram of a cat domain model. In the top layer, you'll notice the interfaces Roarable, Cat, Purrable, and Meowler. These interfaces provide behavior for any derived class that wants to pick and choose (mixin) common behaviors.

The concrete implementation classes are Lion, Tiger, Cheetah, and HouseCat. You will notice that the main interface Cat contains abstract methods getCatKind() and getFurDescription(), and default methods walk(), eat(), and sleep(). As usual, the Java compiler will enforce classes implementing the Cat interface to implement the abstract methods getCatKind() and getFurDescription().

The default methods on the Cat interface have implementation code that serves as functionality for all cats. These default methods will provide behavior to the Cat interface while also centralizing functionality (implementation) for all derived classes. I created the empty (marker) interfaces Roarable, Purrable, and Meowler, which contain respective default methods roar(), purr(), and meow(). These interfaces are similar to the idea of a *mixin*, which in object-oriented languages allows developers to extend or add default behavior to any derived class or interface. Listing 4-22 is a repeat of the beginning of the main class from Listing 4-21, which starts off by creating three cat objects: Tiger, Cheetah, and HouseCat.

Listing 4-22. Creating Three Types of Cat Instances

```

Tiger bigCat = new Tiger();
Cheetah mediumCat = new Cheetah();
HouseCat smallCat = new HouseCat();

```

After creating the cat objects, the code begins by exercising the default and abstract methods. It invokes all of the methods on each cat object. Listing 4-23 shows a cat of type Tiger with all of its methods being invoked.

Listing 4-23. All Methods Invoked on a Tiger Instance

```

System.out.printf("%s with %s fur.\n", bigCat.getCatKind(), bigCat.getFurDescription());
bigCat.eat();
bigCat.sleep();

```

```
bigCat.walk();
bigCat.roar();
bigCat.growl();
System.out.println("-----");
```

Since the rest of the cat kinds have similar tasks, I won't discuss them any further as I believe you have a handle on things. If you've gotten this far, you're a real trooper. Whenever I learn a new skill or concept, I'm always reminded of the programming proverb, "If all you have is a hammer, everything looks like a nail." Getting to learn new language features can often be difficult, especially when you've solved problems for so long without using these new and powerful concepts in Java.

I've only scratched the surface, so I encourage you to dig deeper into the features of Java 8. Now that you've been able to equip yourself with new-found powers of lambdas, let's get back on track with JavaFX.

Properties and Binding

Properties are basically wrapper objects for JavaFX-based object attributes such as String or Integer. Properties allow developers to add listener code to respond when the wrapped value of an object has changed or is flagged as invalid. Also, property objects can be bound to one another. Binding behavior allows properties to update or synchronize their values based on a changed value from another property.

UI Patterns

Before discussing JavaFX's properties and bindings APIs, I would like to share with you a little bit about UI patterns. When developing GUI applications, you will inevitably encounter UI architectural framework concepts such as model view controller ([MVC](#)), presentation model ([PM](#)), model view presenter ([MVP](#)), or model view view-model ([MVVM](#)).

Depending on who you talk to, you might get different explanations; however, these concepts all address the issue of how best to handle synchronization between the model and view. What this means is that when a user interacts (input) with the UI (view) the underlying backend data store (the model) is automatically updated, and vice versa.

Without trying to oversimplify the concepts, I refer you to the actual UI patterns that are involved. The main UI patterns involved are the *Supervising Controller*, *Presentation Model*, and *Mediator*. If you are interested in understanding more about UI patterns, read "GUI Architectures" by Martin Fowler at <http://martinfowler.com/eaaDev/uiArchs.html>.

Because these patterns have been heavily discussed over the years, the JavaFX team has designed and implemented APIs to overcome problems that arose with these UI scenarios. In this section, you will be learning how to use JavaFX properties and bindings to synchronize between your GUI and your data objects. Of course, I can't possibly cover all of the use-case scenarios involving the UI patterns just mentioned, but hopefully I can give you the basics to get you on the right path.

Properties

Before the JavaFX properties API, Java Swing developers adhered to the JavaBean convention (specification), which specifies that objects will contain privately scoped instance variables that have associated getter (accessor) and setter (mutator) methods. For instance, a User class might have a private instance variable password of type String. The associated getter and setter would be the getPassword() and setPassword() methods, respectively. Listing 4-24 shows a User class that follows the older JavaBean convention.

Listing 4-24. A Simple JavaBean Class with Getter and Setter Methods

```
public class User {
    private String password;
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

As a developer who follows the JavaBean convention, you will quickly see the naming convention for the property, which is based on the getter and setter method names and not on the private variable name. In other words, if the getter method were named `getPwd()`, the property would be named `pwd` and would have nothing to do with the attribute named `password`.

The JavaBeans specification also provided an API that has the concept of property change support (`java.beans.PropertyChangeSupport`), which allowed the developer to add handler (listener) code when the property changed. At the time, this solved only part of the issue of property change support. The JavaBeans API started showing its age when it came to binding variables and working with the Collections API. Although the JavaBeans API became a standard way to build domain objects, it still lacked robust ways to synchronize the domain models and views within a GUI application, as described earlier.

Through the years, the JavaBean specification and API gave rise to many third-party tools and frameworks to ease the developer experience. However, wiring up GUI components to JavaBeans using unidirectional or bidirectional binding still got pretty complicated. At times, developers would have resources that were not properly released, which led to object leaks. Developers quickly realized they needed a better way to bind and unbind GUI controls to be wired up to properties.

Types of JavaFX Properties

Let's fast forward to JavaFX's Properties API to see how it handles the common issues. I'll first discuss the different types of JavaFX-based properties. There are two types to be concerned about:

- Read/writable
- Read-only

In short, JavaFX's properties are wrapper objects that hold actual values while providing change support, invalidation support, and binding capabilities. I will address binding later, but for now let's examine commonly used property classes.

Properties are wrapper objects that have the ability to make values accessible as read/writable or read-only. All wrapper property classes are located in the `javafx.beans.property.*` package namespace. Listed here are commonly used property classes. To see all of the property classes, refer to the documentation in Javadoc.

- `javafx.beans.property.SimpleBooleanProperty`
- `javafx.beans.property.ReadOnlyBooleanWrapper`
- `javafx.beans.property.SimpleIntegerProperty`
- `javafx.beans.property.ReadOnlyIntegerWrapper`
- `javafx.beans.property.SimpleDoubleProperty`
- `javafx.beans.property.ReadOnlyDoubleWrapper`

- `javafx.beans.property.SimpleStringProperty`
- `javafx.beans.property.ReadOnlyStringWrapper`

The properties that have a prefix of Simple and a suffix of Property are the read/writable property classes, and the classes with a prefix of ReadOnly and a suffix of Wrapper are the read-only properties. Later, you will see how to create a JavaFX bean using these commonly used properties, but for now let's examine read/writable properties.

Read/Writable Properties

Read/writable properties are, as the name suggests, property values that can be both read and modified. As an example, let's look at JavaFX string properties. To create a string property that is capable of both readable and writable access to the wrapped value, you will use the `javafx.beans.property.SimpleStringProperty` class. Listing 4-25 is a code snippet that demonstrates an instance of a `SimpleStringProperty` class and modifies the property via the `set()` method.

Listing 4-25. Creating a StringProperty Instance

```
StringProperty password = new SimpleStringProperty("password1");
password.set("1234");
System.out.println("Modified StringProperty " + password.get() ); // 1234
```

Here, a declared variable `password` of type `StringProperty` is assigned to an instance of a `SimpleStringProperty` class. It's always a good idea when declaring variables to be more abstract in object-oriented languages. Thus, referencing a `StringProperty` exposes fewer methods of the implementation class (`SimpleStringProperty`). Also notice that the actual value is the string “`password1`”, which is passed into the constructor of the `SimpleStringProperty` class. You will later discover other convenient constructor methods when working with JavaBeans and property objects.

In the case of reading the value back, you would invoke the `get()` method (or `getValue()`), which returns the actual wrapped value(`String`) to the caller. To modify the value, you simply call the `set()` method (or `setValue()`) by passing in a string.

Read-Only Properties

To make a property read-only you use the wrapper classes that are prefixed with `ReadOnly` from the `javafx.beans.property.*` package. To create a property to be read-only, you need to take two steps. First, you instantiate a read-only wrapper class and invoke the method `getReadOnlyProperty()` to return a true read-only property object. Listing 4-26 creates a read-only string property.

Listing 4-26. Creating a Read-Only String Property

```
ReadOnlyStringWrapper userName = new ReadOnlyStringWrapper("jamestkirk");
ReadOnlyStringProperty readOnlyUserName = userName.getReadOnlyProperty();
```

This code snippet actually takes two steps to obtain a read-only string property. You will notice the call to `getReadOnlyProperty()`, which returns a read-only copy (synchronized) of the property. According to the Javadoc API, the `ReadOnlyStringWrapper` class “creates two properties that are synchronized. One property is read-only and can be passed to external users. The other property is readable and writable and should be used internally only.” Knowing that the other property can be read and written could allow a malicious developer to cast the object to type `StringProperty`, which then could be modified at will. From a security perspective, you should be aware of the proper steps to create a true read-only property.

JavaFX JavaBean

Now that you've seen the JavaBean specification's approach to creating domain objects, I will rewrite the earlier JavaBean User class to use JavaFX properties instead. In the process of rewriting the bean, I also wanted to add a read-only property called `userName` to demonstrate the read-only property behavior. Listing 4-27 shows the User class rewritten to use JavaFX properties.

Listing 4-27. The File User.java Rewritten as a JavaFX Bean

```
import javafx.beans.property.*;
public class User {
    private final static String USERNAME_PROP_NAME = "userName";
    private final ReadOnlyStringWrapper userName;

    private final static String PASSWORD_PROP_NAME = "password";
    private final StringProperty password;

    public User() {
        userName = new ReadOnlyStringWrapper(this,
            USERNAME_PROP_NAME, System.getProperty("user.name"));

        password = new SimpleStringProperty(this, PASSWORD_PROP_NAME, "");
    }

    public final String getUserName() {
        return userName.get();
    }
    public ReadOnlyStringProperty userNameProperty() {
        return userName.getReadOnlyProperty();
    }
    public final String getPassword() {
        return password.get();
    }
    public final void setPassword(String password) {
        this.password.set(password);
    }
    public StringProperty passwordProperty() {
        return password;
    }
}
```

You may notice some obvious differences in the way I instantiated the `ReadOnlyStringWrapper` and `SimpleStringProperty` classes. Similar to the JavaBean property change support, JavaFX properties have constructors that will allow you to specify the bean itself, its property name, and its value. As a simple example, Listing 4-28 shows the instantiations of a read-only and a read-writable property using the JavaFX property change support-based constructors. The `userName` variable is assigned a read-only property, and the `password` variable is assigned a read/writable property.

Listing 4-28. Instantiating a Read-Only and a Read-Writable Property

```
userName = new ReadOnlyStringWrapper(this, USERNAME_PROP_NAME, System.getProperty("user.name"));
password = new SimpleStringProperty(this, PASSWORD_PROP_NAME, "");
```

It is good practice to use the property change support-based constructors for many reasons, such as third-party tooling and testing. But most of all, when dealing with property change support, you need access to the bean and its other properties.

One last thing to mention is that, in Listing 4-26, notice that the getter and setter methods are final. Making the getter and setter final prevents any derived classes from overriding and possibly changing the underlying property.

Property Change Support

Property change support is the ability to add handler code that will respond when a property changes. JavaFX property objects contain an `addListener()` method. This method will accept two types of functional interfaces, `ChangeListener` and `InvalidationListener`. Recall that functional interfaces are single abstract methods that are expressed using the Java lambda syntax. All JavaFX properties are descendants of the `ObservableValue` and `Observable` interfaces (method overloading), which provide the `addListener()` methods for `ChangeListener` and `InvalidationListener`, respectively. Finally, remember that it is important to clean up listeners by removing them. The purpose of cleaning up listeners is to reclaim memory. To remove them, you will invoke the `removeListener()` method by passing into it a referenced (named) listener as opposed to an anonymous inner class or anonymous lambda expression.

Listing 4-29 shows how to create a `ChangeListener` to be registered with a property. As the property's value changes, the `change()` method will be invoked. I provided the implementations for both the anonymous inner class and lambda expression-style syntax for you to compare.

Listing 4-29. Adding a ChangeListener Instance to Respond When the Property Value Changes

```
SimpleIntegerProperty xProperty = new SimpleIntegerProperty(0);

// Adding a change listener (anonymous inner class)
xProperty.addListener(new ChangeListener<Number>(){
    @Override
    public void changed(ObservableValue<? extends Number> ov,
                        Number oldValue, Number newValue) {
        // code goes here
    }
});

// Adding a change listener (lambda expression)
xProperty.addListener((ObservableValue<? extends Number> ov,
                       Number oldValue, Number newValue) -> {
    // code goes here
});
```

Listing 4-30 shows how to create an `InvalidationListener` to be registered with a property. As the property's value changes, the `invalidated()` method will be invoked. I've provided the implementations for both the anonymous inner class and lambda expression-style syntax for you to compare.

Listing 4-30. Adding an `InvalidationListener` Instance to Respond When the Property Value Changes

```
// Adding a invalidation listener (anonymous inner class)
xProperty.addListener(new InvalidationListener() {
    @Override
    public void invalidated(Observable o) {
        // code goes here
    }
});

// Adding a invalidation listener (lambda expression)
xProperty.addListener((Observable o) -> {
    // code goes here
});
```

Often developers become confused about the difference between a `ChangeListener` and an `InvalidationListener`. In Listing 4-30, you'll notice that using a `ChangeListener` you will get the `Observable` (`ObservableValue`), the old value, and the new value, while using the `InvalidationListener` only gets the `Observable` object (property).

In the Javadoc documentation, the difference between a `ChangeListener` and `InvalidationListener` is described as follows:

“A change event indicates that the value has changed. An invalidation event is generated, if the current value is not valid anymore. This distinction becomes important, if the `ObservableValue` supports lazy evaluation, because for a lazily evaluated value one does not know if an invalid value really has changed until it is recomputed. For this reason, generating change events requires eager evaluation while invalidation events can be generated for eager and lazy implementations.”

The `InvalidationListener` provides a way to mark values as invalid but does not recompute the value until it is needed. This is often used in UI layouts or custom controls, where you can avoid unnecessary computations when nodes don't need to be redrawn/repositioned during a layout request or draw cycle. When using the `ChangeListener`, you normally want eager evaluation such as the validation of properties on a form-type application. That doesn't mean you can't use `InvalidationListeners` for validation of properties; it just depends on your performance requirements and exactly when you need the new value to be recomputed (evaluated). When you access the observable value, it causes the `InvalidationListener` to be eager.

Binding

Simply put, binding has the idea of at least two values (properties) being synchronized. This means that when a dependent variable changes, the other variable changes. JavaFX provides many binding options that enable the developer to synchronize between properties in domain objects and GUI controls. In this section, you learn about three binding strategies when binding property objects.

This section focuses on the following JavaFX API packages:

- `javafx.beans.binding.*`
- `javafx.beans.property.*`

Binding properties is quite easy to do. The only requirement is that the property invoking the bind must be a read/writeable property. To bind a property to another property, you will invoke the `bind()` method. This method will bind in one direction (unidirectional). For instance, when property A binds to property B, the change in property B will update property A, but not the other way. If A is bound to B you can't update A, as you'll get `RuntimeException`: A bound value cannot be set.

The following are three additional binding strategies to consider using in JavaFX's Properties API:

- Bidirectional binding on a Java Bean
- High-level binding using the Fluent API
- Low-level binding using `javafx.beans.binding.*` binding objects

Bidirectional Binding

Bidirectional binding allows you to bind properties with the same type allowing changes on either end while keeping a value synchronized. When binding bi-directionally, it's required that both properties must be read/writable. It's pretty easy to bind two properties bidirectionally through the use of the `bindBidirectional()` method. To demonstrate, I've created a simple example.

Listing 4-31 is a Contact bean having a property `firstName` that is bound bidirectionally to a local variable `fname` of type `StringProperty`.

Listing 4-31. Bidirectional Binding Between the `firstName` Property and a Regular String Property Variable

```
Contact contact = new Contact("John", "Doe");
StringProperty fname = new SimpleStringProperty();

fname.bindBidirectional(contact.firstNameProperty());

contact.firstNameProperty().set("Play");
fname.set("Jane");

System.out.println("contact.firstNameProperty = " + contact.firstNameProperty().get() );
System.out.println("fname = " + fname.get() );
```

The output of this code snippet is as follows:

```
contact.firstNameProperty = Jane
fname = Jane
```

High-Level Binding

Introduced in JavaFX 2.0 is a fluent interface API to bind properties. The fluent APIs are methods that allow you to perform operations on properties using English-like method names. For example, if you have a numeric property, there are methods like `multiply()`, `divide()`, `subtract()`, and so on. Another example would be a string property having methods like `isEqualTo()`, `isNotEqualTo()`, `concat()`, and similar. As an example, Listing 4-32 shows how to create a property that represents the formula for the area of a rectangle.

Listing 4-32. Creating a Property (area) Using the High-Level Binding Strategy for Calculating the Area of a Rectangle

```
// Area = width * height
IntegerProperty width = new SimpleIntegerProperty(10);
IntegerProperty height = new SimpleIntegerProperty(10);
NumberBinding area = width.multiply(height);
```

This code demonstrates high-level binding by using the fluent API from the `javafx.beans.binding.IntegerExpression` parent interface. This example binds by using the `multiply()` method, which returns a `NumberBinding` containing the computed value. What's nice is that the binding is lazy-evaluated, which means the computation (multiplying) doesn't occur unless you invoke the property's (`area`) value via the `get()` (or `getValue()`) method. For all available fluent interfaces, see the Javadoc for `javafx.beans.binding.*` packages.

Low-Level Binding

When using low-level binding, you would use a derived `NumberBinding` class, such as a `DoubleBinding` class for values of type `Double`. With a `DoubleBinding` class, you will override its `computeValue()` method so that you can use the familiar operators such as `*` and `-` to formulate complex math equations. The difference between high-and low-level binding is that high-level uses methods such as `multiply()` and `subtract()` instead of the operators `*` and `-`. Listing 4-33 shows how to create a low-level binding for the formula for the volume of a sphere.

Listing 4-33. A Property Binding (`volumeOfSphere`) Using the Low-Level Binding Strategy

```
DoubleProperty radius = new SimpleDoubleProperty(2);

DoubleBinding volumeOfSphere = new DoubleBinding() {
    {
        super.bind(radius); // initial bind
    }

    @Override
    protected double computeValue() {
        // Math.pow() (power) cubes the radius
        return (4 / 3 * Math.PI * Math.pow(radius.get(), 3));
    }
};
```

If you've gotten this far, you are definitely ready to begin building GUIs. So far you've had a chance to learn about lambda expressions, properties, and bindings. Next, I want to close out this chapter with an example showing how to build a snazzy-looking logon dialog that uses shapes, lambda expressions, properties, and bindings (with a hint of UI controls).

A Logon Dialog Example

Logon dialogs are typical of any application when authenticating users. In this example, I created a simple form application that will involve all of the skills you've learned from Chapter 3 and in this chapter thus far. Before getting deep into the code, which you'll see in Listing 4-34, let's look at the logon dialog window. Figure 4-3 shows the initial display of the logon dialog window that the code will create.

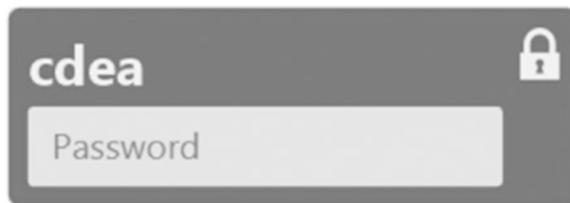


Figure 4-3. Initial display of the logon dialog window

The following are the instructions for interacting with the application.

1. The user will have three attempts to enter the correct password. Figure 4-4 shows the user making the first attempt to log on. (Note: The password is password1.)

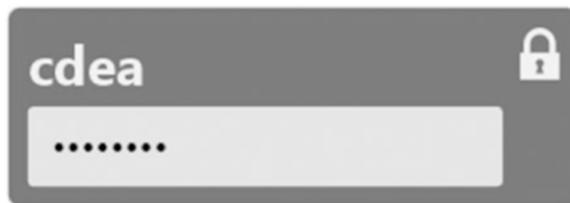


Figure 4-4. The user is entering the password into the logon dialog window

2. As users enter the password, they have an opportunity to press the Enter key. If the password is incorrect, a red X appears to the right of the password text field. Figure 4-5 shows an invalid logon.

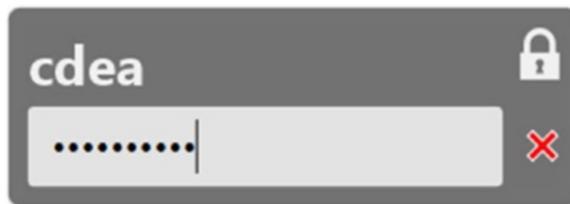


Figure 4-5. The user has pressed Enter with an invalid password

3. If the user types the correct password in real-time, a green check appears. After the green check appears, the user may press the Enter key to be granted access. Figure 4-6 shows the user entering a valid password.



Figure 4-6. The user has entered a valid password

Login Dialog Source Code

Listing 4-34 shows the source code for the JavaFX example logon dialog application. Listing 4-35 lists the style.css file to style this JavaFX example logon dialog application.

Listing 4-34. The Java Source Code for a JavaFX Example Logon Dialog Application

```
/*
 * A login form to demonstrate lambdas, properties and bindings.
 * @author cdea
 */
public class FormValidation extends Application {

    private final static String MY_PASS = "password1";
    private final static BooleanProperty GRANTED_ACCESS = new SimpleBooleanProperty();
    private final static int MAX_ATTEMPTS = 3;
    private final IntegerProperty ATTEMPTS = new SimpleIntegerProperty();

    @Override
    public void start(Stage primaryStage) {
        // create a domain model representing a user
        User user = new User();

        // create a transparent stage
        primaryStage.initStyle(StageStyle.TRANSPARENT);
        primaryStage.setAlwaysOnTop(true);

        Group root = new Group();
        Scene scene = new Scene(root, 320, 112, null);

        // load style.css to style JavaFX nodes
        scene.getStylesheets()
            .add(getClass()
                .getResource("/style.css")
                .toExternalForm());

        primaryStage.setScene(scene);
    }
}
```

```
// rounded rectangular background
Rectangle background = new Rectangle();
background.setId("background-rect");

background.widthProperty()
    .bind(scene.widthProperty()
        .subtract(5));
background.heightProperty()
    .bind(scene.heightProperty()
        .subtract(5));

// a read only field holding the user name.
Label userName = new Label();
userName.setId("username");
userName.textProperty()
    .bind(user.userNameProperty());

HBox userNameCell = new HBox();
userNameCell.getChildren()
    .add(userName);

// When Label's text is wider than the background minus the padlock icon.
userNameCell.maxWidthProperty()
    .bind(primaryStage.widthProperty()
        .subtract(45));
userNameCell.prefWidthProperty()
    .bind(primaryStage.widthProperty()
        .subtract(45));

// padlock
Region padlock = new Region();
padlock.setId("padlock");

HBox padLockCell = new HBox();
padLockCell.setId("padLockCell");
HBox.setHgrow(padLockCell, Priority.ALWAYS);
padLockCell.getChildren().add(padlock);

// first row
HBox row1 = new HBox();
row1.getChildren()
    .addAll(userNameCell, padLockCell);

// password text field
PasswordField passwordField = new PasswordField();
passwordField.setId("password-field");
passwordField.setPromptText("Password");
passwordField.prefWidthProperty()
    .bind(primaryStage.widthProperty()
        .subtract(55));
```

```

// populate user object's password from password field
user.passwordProperty()
    .bind(passwordField.textProperty());

// error icon
Region deniedIcon = new Region();
deniedIcon.setId("denied-icon");
deniedIcon.setVisible(false);

// granted icon
Region grantedIcon = new Region();
grantedIcon.setId("granted-icon");
grantedIcon.visibleProperty()
    .bind(GRANTED_ACCESS);

// hide and show denied icon and granted icon
StackPane accessIndicator = new StackPane();
accessIndicator.getChildren().addAll(deniedIcon, grantedIcon);

// second row
HBox row2 = new HBox(3);
row2.getChildren().addAll(passwordField, accessIndicator);
HBox.setHgrow(accessIndicator, Priority.ALWAYS);

// user hits the enter key on the password field
passwordField.setOnAction(actionEvent -> {
    if (GRANTED_ACCESS.get()) {
        System.out.printf("User %s is granted access.\n",
            user.getUserName());
        System.out.printf("User %s entered the password: %s\n",
            user.getUserName(), user.getPassword());
        Platform.exit();
    } else {
        deniedIcon.setVisible(true);
        ATTEMPTS.set(ATTEMPTS.add(1).get());
    }
});

// listener when the user types into the password field
passwordField.textProperty().addListener((obs, ov, nv) -> {
    GRANTED_ACCESS.set(passwordField.getText().equals(MY_PASS));
    if (GRANTED_ACCESS.get()) {
        deniedIcon.setVisible(false);
    }
});

// listener on number of attempts
ATTEMPTS.addListener((obs, ov, nv) -> {
    // failed attempts
    System.out.println("Attempts: " + ATTEMPTS.get());
});

```

```

        if (MAX_ATTEMPTS == nv.intValue()) {
            System.out.printf("User %s is denied access.\n", user.getUserName());
            Platform.exit();
        }
    });

VBox formLayout = new VBox(4);
formLayout.getChildren().addAll(row1, row2);
formLayout.setLayoutX(12);
formLayout.setLayoutY(12);

root.getChildren().addAll(background, formLayout);

primaryStage.show();
}

public static void main(String[] args) {
    launch(args);
}
}
}

```

Listing 4-35. The style.css file to Style the JavaFX Example Logon Dialog Application

```

.root {
    common-foreground-color: rgb(255, 255, 255, 0.90);
}

#background-rect {
    -fx-translate-x: 5px;
    -fx-translate-y: 5px;
    -fx-arc-height: 15;
    -fx-arc-width: 15;
    -fx-fill: rgba(0, 0, 0, .55);
    -fx-stroke: common-foreground-color;
    -fx-stroke-width: 1.5;
}

#username {
    -fx-font-family: "Helvetica";
    -fx-font-weight: bold;
    -fx-font-size: 30;
    -fx-text-fill: common-foreground-color;
    -fx-smooth: true;
}

#padLockCell {
    -fx-alignment: center-right;
}

```

```

#padlock {
    -fx-position-shape: true;
    -fx-padding: 0 0 0 20;
    -fx-scale-shape: false;
    -fx-background-color: common-foreground-color;
    -fx-shape: "M24.875,15.334v-4.876c0-4.894-3.981-8.875-8.875s-8.875,3.981-8.875,
                8.875v4.876H5.042v15.083h21.916V15.334H24.875zM10.625,10.458c0-2.964,
                2.411-5.375,5.375s5.375,2.411,5.375,5.375v4.876h-10.75V10.458zM18.272,
                26.956h-4.545l1.222-3.667c-0.782-0.389-1.324-1.188-1.324-2.119c0-1.312,
                1.063-2.375,2.375s2.375,1.062,2.375,2.375c0,0.932-0.542,1.73-1.324,
                2.119L18.272,26.956z";
}

#denied-icon {
    -fx-position-shape: true;
    -fx-padding: 0 0 0 20;
    -fx-scale-shape: false;
    -fx-border-color: white;
    -fx-background-color: rgba(255, 0, 0, .9);
    -fx-shape: "M24.778,21.419 19.276,15.917 24.777,10.415 21.949,7.585 16.447,13.087 10.945,
                7.585 8.117,10.415 13.618,15.917 8.116,21.419 10.946,24.248 16.447,
                18.746 21.948,24.248z";
}

#granted-icon {
    -fx-position-shape: true;
    -fx-padding: 0 0 0 20;
    -fx-scale-shape: false;
    -fx-border-color: white;
    -fx-background-color: rgba(0, 255, 0, .9);
    -fx-shape: "M2.379,14.729 5.208,11.899 12.958,19.648 25.877,6.733 28.707,
                9.561 12.958,25.308z";
}

#password-field {
    -fx-font-family: "Helvetica";
    -fx-font-size: 20;
    -fx-text-fill:black;
    -fx-prompt-text-fill:gray;
    -fx-highlight-text-fill:black;
    -fx-highlight-fill: gray;
    -fx-background-color: rgba(255, 255, 255, .80);
}

```

Explanation of the Code

The JavaFX application in Listing 4-34 simulates a logon dialog window allowing a user to enter a valid password. The code mainly demonstrates binding JavaFX JavaBeans properties with UI controls by reusing the User class that was shown earlier, in Listing 4-27.

Before you begin the code walkthrough, I want to mention the difference between Listing 4-34 and the prior book's second edition in Chapter 3. The main difference from the older code example was that it was entirely in Java code as opposed to Java code and JavaFX CSS styling. In this chapter, you will see this example separated in two files—one in Listing 4-34 (Java code) and the other in Listing 4-35 (JavaFX CSS file). I did this on purpose to allow you to focus on the code related to topics in this chapter such as lambdas, properties, etc. If you don't understand the [JavaFX CSS](#) styling applied to UI controls, don't worry, as I will explain this in detail in Chapter 15 Custom User Interfaces.

So, let's begin walking through this login dialog example by describing the class member variables and then look at the body of the code.

Class Member Variables

I'll begin by describing the class variables. The variable `MY_PASS` contains a hardcoded string of the password of "password1". Next, the `GRANTED_ACCESS` variable is declared of type `SimpleBooleanProperty` class, which by default has a value of `false`. This variable will be later bound to the green checkmark (`grantedIcon`) icon node's `visible` property. Basically, as the user enters the correct keystrokes for the password, the `GRANTED_ACCESS` will become `true`, which will trigger the setting of the green checkmark icon node's `visible` property to `true`, thus displaying the icon in the scene graph. When the password is incorrect, the `visible` property becomes `false`, which hides the green checkmark node (`Region`). The constant variable `MAX_ATTEMPTS` is the maximum number of login attempts the users can try. Lastly, the `ATTEMPTS` variable of type `IntegerProperty` holds the current number of login attempts the user has made.

The `start()` Method

The `start()` method is where the JavaFX application thread's lifecycle begins setting up the stage, scene, and children nodes to be displayed. In the `start()` method, a `User` object is created to be used as the domain object to be synchronized with the GUI form. To see the code for the `User` class, refer to Listing 4-26. Next, the stage is set to be transparent by invoking the `Stage.initStyle(StageStyle.TRANSPARENT)` method. This will allow you to create translucent and irregularly shaped windows without the decoration of the native OS title bar and borders surrounding the main content. You'll also notice the statement `setAlwaysOnTop(true)`, which will cause the Stage (dialog window) to be on top of all other application windows on the desktop. For convenience, the following code is repeated, showing you how to enable your Stage window to be transparent and to appear on top of all other application windows.

```
primaryStage.initStyle(StageStyle.TRANSPARENT);
primaryStage.setAlwaysOnTop(true);
```

After the code initializes the stage, a `Scene` object is created with a root node of type `Group` that will allow children nodes to be added. A JavaFX `Group` node is a simple fixed size parent container. Later in the book, you will learn about other layout parent containers that are capable of resizing dynamically. For now, the `Group` node is used for this simple example that has very few children nodes.

Back to the root node, even though the root node is transparent, a rectangular background will need to be created to act as the surface to put the controls on top of. The rectangular background will need to be the same size as the defined `Scene` (logon screen) object. The following code snippet is the creation of a `Scene` object having a `Group` node as its root node with a background fill color of null, thus making the window transparent.

```
Scene scene = new Scene(root, 320, 112, null);
```

The rectangular background of the logon screen is actually a rounded rectangle ([Rectangle](#)) having an arc height and width of 15 pixels with a fill color of black with an opacity level of 55%. All styling information is located in the file `style.css` shown in Listing 4-35. To apply JavaFX CSS, the code will load a CSS file located in your application's resources directory. The following code snippet demonstrates loading and applying the stylesheet definitions to the scene.

```
// load style.css to style JavaFX nodes
scene.getStylesheets()
    .add(getClass()
        .getResource("/style.css")
        .toExternalForm());
```

To give things a consistent look, the code declares a `common-foreground-color` CSS variable, containing a color of white with an opacity of 90% percent. To see the full `style.css` file's CSS styling definitions, check out the earlier mentioned source code from Listing 4-35.

Continuing with Listing 4-34, inside the `start()` method after setting up the initial rectangular background surface, the code creates a [Label](#) node that will represent the user's username as a read-only field (per the `User` class). To obtain the user's real username, the code calls Java's `System.getProperty("user.name")` method. The label node is set up with a 30 point Helvetica font having a fill color of the `common-foreground-color` (white) attribute. The label node's text property is bound to the `User` object's read-only `userName` property using the `bind()` method. A JavaFX `Label` node was used to represent the user's username field. The field can tend to be wider than the width of the parent container's maximum width due to a long text string. Because of this, when a JavaFX `Label`'s text length exceeds the allowed width, the text on the label will be shortened and end with an ellipsis.

Next, the code creates an `HBox` layout pane node that will hold the `Label` node (`username`). The `HBox` (`userNameCell`) will be binding its max and preferred width property by using the JavaFX properties fluent API to ensure that it doesn't hog up all the horizontal space and allow the padlock icon to have room to be placed beside it. Figure 4-3 shows the padlock icon in the upper-right corner of the logon dialog window. The following code snippet shows an `HBox` containing the `TextNode` (`username`) from Listing 4-34.

```
// wrap text node
HBox userNameCell = new HBox();
userNameCell.getChildren().add(username);

userNameCell.prefWidthProperty()
    .bind(primaryStage.widthProperty()
        .subtract(45));
```

I've mentioned very little about the `HBox` layout control here, because it will be discussed further in Chapter 5, which covers layouts and the Scene Builder tool. In the prior edition of this book, I created the padlock icon using a JavaFX `SVGPath` node; however, to separate nodes from their styling definitions, you have to use a `Region` node instead of the `SVGPath` node. When using an `SVGPath` node and specifying its SVG path notation text, you will find out that it can only be done programmatically (Java code) by using the `setContent()` method. In order to mimic an `SVGPath` node, but have the SVG path notation code reside in the CSS styling file, you will have to employ the `Region` node.

A `Region` node is a very versatile node that is typically used as a base class for many resizable parent container nodes used in JavaFX layout nodes. The `Region` node has a plethora of CSS styling attributes that enable you to not only style borders and backgrounds, but interestingly enough also specify an SVG path notation to create custom shapes. Looking at Listing 4-34 you will see the SVG path notation specified to draw the various shaped icons. When specifying the SVG path notation as a value, you will notice the `-fx-shape` attribute being used.

Recall that Chapter 3 covered generating complex shapes using path elements to create an ice cream cone, as depicted in Figure 3-10. Having said this, the Region class is very similar to the Path node, except that the child path elements are expressed in W3C SVG path notation as a string. To render a path, the string representing the path notation is defined in the style.css file. To see how to use SVG paths, visit <http://www.w3.org/Graphics/SVG>. To create the padlock icon, I originally had gone to the web site of the JavaScript library called *Raphaël* by Dmitry Baranovskiy; however, the site is no longer available. At the time, I was able to obtain very nice SVG icons to be used in this example. Next, the code creates an HBox as a container for the first row (row1) to hold the userNameCell and padLock nodes.

After creating the first row, the code proceeds by creating the PasswordField UI control. Like the Label node, the PasswordField UI control's font can be set programmatically or by using JavaFX CSS. The prompt text is shown in Figure 4-3 of the initial logon dialog with the prompt text as "Password."

For some final styling on the password field, the code continues to use JavaFX CSS styling, which you will get a chance to see in later chapters. The CSS style applied to the password field is a white background with an opacity of 80 percent. Still working with the password field, the code uses the fluent interface subtract() method to bind the preferred width that subtracts 55px (pixels). This allows room for the StackPane that contains the granted and denied icons. Last, the user object's passwordProperty is bound to the PasswordField's textProperty. This is a unidirectional bind, which means that as the user enters text into the UI password field, the User object's password property changes and not the other way around.

Next, the code creates two more icons using the Region node that represents an 'X' and a checkmark, denoting denied (deniedIcon) and granted (grantedIcon) access icons, respectively. These icons will be placed in a StackPane layout node. The StackPane layout allows child nodes to be stacked, hence the name. In this scenario the deniedIcon and grantedIcon nodes are stacked on top of each other. This trick allows you to toggle between icons using the setVisible() method. As a whole (stack pane) node, it will be positioned to the right of the password field. As the user types into the password field, the Boolean property GRANTED_ACCESS is updated, which updates the grantedIcon's visible property. The following code statement shows the visible property bound to the GRANTED_ACCESS property.

```
grantedIcon.visibleProperty().bind(GRANTED_ACCESS);
```

Completing the second row, the code creates another HBox similar to the first row, this time holding the password field and the StackPane node containing the icons (denied and granted icons). Next, the code continues by wiring up the UI controls and adding property change listeners (ChangeListeners) to various properties. The code proceeds with the passwordField by adding an action as a functional interface for the method setOnAction() in response to the user pressing the Enter key. The following handler code is set on the password field when the user presses the Enter key.

```
passwordField.setOnAction(actionEvent -> {
    if (GRANTED_ACCESS.get()) {
        System.out.printf("User %s is granted access.\n",
            user.getUserName());
        System.out.printf("User %s entered the password: %s\n",
            user.getUserName(), user.getPassword());
        Platform.exit();
    } else {
        deniedIcon.setVisible(true);
        ATTEMPTS.set(ATTEMPTS.add(1).get());
    }
});
```

Essentially, if the password matches and the user presses Enter, the program exits and outputs the following:

```
Attempts: 1
User cdea is granted access.
User cdea entered the password: password1
```

You'll also notice that the code increments the ATTEMPTS property by invoking the fluent interface method `add()`.

Still working with the password field, the code adds a change listener to listen for keystrokes from the user. As the user types into the password field, the password is compared against the stored (hardcoded) password. If the password is valid, the code sets the GRANTED_ACCESS property to true and hides the deniedIcon icon via the `setVisible(false)` method.

Finally, I added the last bit of wiring for when a user has too many failed attempts. I basically added a change listener onto the ATTEMPTS property. The change listener will compare MAX_ATTEMPTS against ATTEMPTS to see if they are equal. If so, it will invoke the JavaFX's `Platform.exit()` method to gracefully exit the application. The rest is assembling the rows into a VBox layout and adding it to the root node for the scene to be shown.

Summary

This chapter covered a lot of ground with properties and bindings, especially relating to new Java 8 language features. You were introduced to the new features in Java 8, such as lambda expressions, stream APIs, and default methods. Starting with Java lambdas, you were able to learn the syntax and use functional interfaces. Next, you were able to use the stream API and common aggregate functions. After an example of using the stream APIs to manipulate elements in a collection, you then learned how to extend behavior on Java interfaces with default methods. Then you explored UI patterns, the Properties API, and the idea of change support. With a solid understanding of working with properties, you then learned about various binding strategies. To wrap up the chapter, you saw an example of how to create a logon dialog window, which employed all the concepts you've learned so far. Next, in Chapter 5, you'll explore JavaFX UI layout and UI controls.

CHAPTER 5



Layouts and Scene Builder

As more and more graphically rich applications such as desktops, smartphones, and tablets devices find their way into our lives, it is very important to design GUI applications that provide better usability across the different form factors. With the increase of the number different screen sizes, a UI developer will need to learn how to create applications with UI layout management in mind. UI layout management is the ability to position or resize child nodes on the JavaFX scene graph. This chapter discusses the most common JavaFX layouts used in many GUI applications.

This chapter also introduces the Scene Builder tool, a graphical (WYSIWYG) editor that allows you to develop UIs graphically without having to manually code (programmatically) UI elements. In this section, you will learn about how to download and install the UI editor tool. Once Scene Builder is installed, you will learn how to build a typical UI form consisting of common UI nodes such as text fields and buttons. Later, you will create an application utilizing the UI form's serialized file format (FXML). Using FXML, you will have an opportunity to learn aspects of the popular MVC (Model/View/Controller) UI pattern through JavaFX's mechanism to wire up UI elements to controller code.

Layouts

One of the greatest challenges in building user interfaces is laying out UI controls onto the display area. In GUI applications, it is ideal to allow the user to resize a viewable area (Window) while UI controls also resize in order to provide a pleasant user experience. Similar to Java's Swing API, the JavaFX API has stock layouts that provide the most common ways to display UI controls onto the scene graph. The following are the JavaFX layouts discussed in this section:

- `javafx.scene.layout.HBox`
- `javafx.scene.layout.VBox`
- `javafx.scene.layout.FlowPane`
- `javafx.scene.layout.BorderPane`
- `javafx.scene.layout.GridPane`

Note To see other stock layouts not discussed in this chapter, refer to the Javadoc documentation at <https://docs.oracle.com/javase/8/javafx/api/javafx/scene/layout/Pane.html>.

HBox

The HBox layout's job is to place JavaFX child nodes in a horizontal row. As child nodes are added consecutively, each is appended to the end (right side). By default, the HBox layout honors the child's preferred width and height. When the parent node is not resizable (a [Group](#) node, for example), the row height of the HBox takes on the height of the child node with the greatest preferred height. Also, by default, each child node aligns to the top-left ([Pos.TOP_LEFT](#)) position.

The HBox can be configured to have resizing ranges to allow the parent to manage and calculate space available upon layout requests. There are many ways you can programmatically alter the HBox's layout constraints, such as border, padding, margin, spacing, and alignment. Although in this chapter you will learn how to set constraints using a programmatic approach, there are also alternative strategies. One well known and preferred strategy is to style or set constraints using JavaFX CSS attributes; you will catch a glimpse of that approach later in this chapter, and more details in Chapter 15, on custom UIs.

When dealing with nonresizable child nodes such as Shape nodes, the parent takes into account the Shape's rectangular bounds (ParentInBounds), its width and height. In contrast, when using resizable nodes such as a [TextField](#) control, the parent (HBox) has the ability to manage and compute the available space for the [TextField](#) to grow horizontally. To grow UI controls horizontally within an HBox, use the static [HBox.setHgrow\(\)](#) method. Also, assume the HBox's parent is resizable and takes up the available space such as a [BorderPane](#)'s center region. The following code snippet sets a [TextField](#) control to grow horizontally when the parent HBox's width is resized:

```
TextField myTextField = new TextField()
HBox.setHgrow(myTextField, Priority.ALWAYS);
```

An HBox Example

Before you look at the code example, let's look at a picture of child nodes inside an HBox node and see how each are spaced appropriately. Figure 5-1 shows an HBox containing four rectangles—r1 through r4. The picture shows the widths of the border, padding, margin, and spacing for the child nodes.

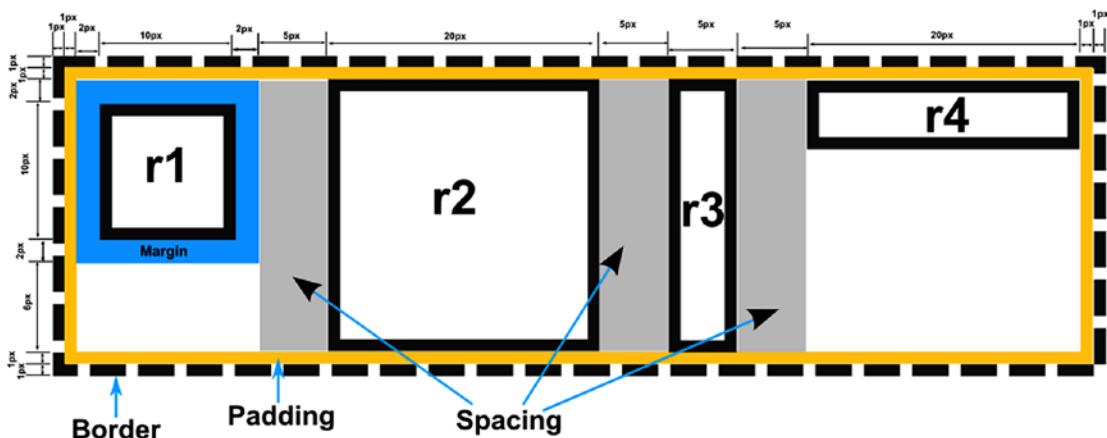


Figure 5-1. An example of an HBox layout containing four rectangles

An example of an HBox layout in action, the code shown in Listing 5-1 uses four rectangles as child nodes with varying HBox constraints. Nonresizable nodes such as rectangle shapes are used in the example to demonstrate the many spacing attributes for the HBox layout control. For brevity, only the relevant code is listed here. To see the full listing, visit the Source Code/Download tab of the book's page at www.apress.com to download the source code.

Listing 5-1. The HBoxExample.java file demonstrates using an HBox layout with shapes as child nodes

```
Group root = new Group();
Scene scene = new Scene(root, 300, 250);

// pixels space between child nodes
HBox hbox = new HBox(5);

// The border is blue, dashed,
// 0% radius for all corners,
// a width of 1 pixel
BorderStroke[] borderStrokes = new BorderStroke[] {
    new BorderStroke(Color.BLUE,
                      BorderStrokeStyle.DASHED,
                      new CornerRadii(0.0, true),
                      new BorderWidths(1.0))
};

hbox.setBorder(new Border(borderStrokes));

// padding between child nodes only
hbox.setPadding(new Insets(1));

// rectangles r1 to r4
Rectangle r1 = new Rectangle(10, 10);
Rectangle r2 = new Rectangle(20, 20);
Rectangle r3 = new Rectangle(5, 20);
Rectangle r4 = new Rectangle(20, 5);

// margin of 2 pixels
HBox.setMargin(r1, new Insets(2,2,2,2));

hbox.getChildren().addAll(r1, r2, r3, r4);

root.getChildren().add(hbox);

// once shown display the dimensions all added up.
primaryStage.setOnShown((WindowEvent we) -> {
    System.out.println("hbox width " + hbox.getBoundsInParent().getWidth());
    System.out.println("hbox height " + hbox.getBoundsInParent().getHeight());
});

primaryStage.setTitle("HBox Example");
primaryStage.setScene(scene);
primaryStage.show();
```

After running the code from Listing 5-1, the following output is shown in Figure 5-2. In Figure 5-2, an HBox is shown surrounded with a blue dashed border. The HBox instance has four varying sized rectangles.

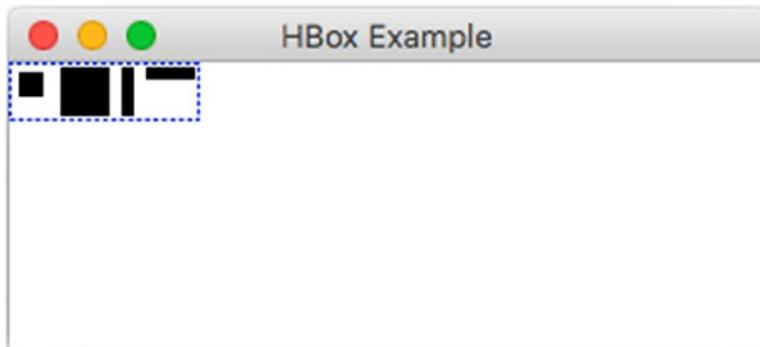


Figure 5-2. The output of the file `HBoxExample.java`, demonstrating the HBox layout

Code Walkthrough

Listing 5-1 begins by instantiating an HBox using the single-argument constructor. This constructor takes a value of type double, which specifies the horizontal space (in pixels) between child nodes. This is also the equivalent of invoking the `setSpacing()` method on the HBox instance. After creating an instance of an HBox with five-pixel spacing, the code creates a blue colored dashed border with a zero radius for all four corners with a one-pixel stroke width. The following code snippet creates an array of one BorderStroke instance to be drawn around the HBox layout pane node.

```
// The border is blue, dashed, 0% radius for all corners,
// a width of 1 pixel
BorderStroke[] borderStrokes = new BorderStroke[] {
    new BorderStroke(Color.BLUE,
                     BorderStrokeStyle.DASHED,
                     new CornerRadii(0.0, true),
                     new BorderWidths(1.0))
};
hbox.setBorder(new Border(borderStrokes));
```

When creating borders, you will remember from Chapter 3 that you used the same concepts to create shapes as when creating strokes. A border's stroke can be created with various styles and colors. All JavaFX layout panes will have a `setBorder()` method allowing you to create borders. To create a border, you have to create an array of `BorderStroke` objects having at least one. This code creates a `BorderStroke` instance with the color blue with a `BorderStrokeStyle.DASHED` value. Other stroke styles are `DOTTED`, `SOLID`, and `NONE`. Next, the corner radii parameter is set with the constructor `CornerRadii(percent, isPercentage)`. When using a 0.0 value, the corners have a zero radius, which creates squared shaped corners. By varying the radius, the rectangle can have rounded corners appearing like a pill shape. Lastly, the stroke width parameter takes an instance of a `BorderWidths` class (double precision).

After the border has been set, the `HBox (hbox)` instance's `setPadding()` method is invoked with an `Insets(1)`, which will set all four sides to use a one-pixel space between the border and child nodes. As you've seen earlier in Figure 5-1, the border and inset spacing show the surrounding perimeter.

Because spacing, padding, margin, and the border pixel widths can often be confusing, I've illustrated an `HBox` in Figure 5-1. The figure displays three gray rectangular areas, each five pixels wide, that represent the spacing between the child nodes (`r1` through `r4`). Setting the spacing uses an `Insets` object that allows you to create space between the `HBox`'s child nodes. Spacing is created from the `HBox`'s constructor or the `setSpacing()` method. The spacing shown in Figure 5-1 is the space between the four rectangles (pictured as gray between the rectangle shapes). To not be confused with padding, you'll notice that the inner part of the `HBox` between the border and the child nodes is the padding (colored in yellow) taking up a one-pixel width surrounding the row.

After creating all four rectangles, the code sets the margin on rectangle `r1`. To set the margin, you'll notice the static method `HBox.setMargin(Node, Insets)` instead of a method on the `HBox` instance. The code creates an `Insets` object with a margin of two pixels for the top, right, bottom, and left. As shown in Figure 5-1, this is the surrounding space around rectangle `r1` (blue). To calculate the size of the `HBox`'s dimensions, you basically add the border width, padding, spacing, margin, and the child nodes' bounds (`ParentInBounds`). When the parent of the `HBox` is a `Group` node (not resizable) and the child nodes aren't resizable, the `HBox`'s dimension will total 78 pixels wide by 24 pixels high. Tables 5-1 and 5-2 show the breakdown of the width and height of the `HBox` instance.

Table 5-1. The `HBox`'s Total Width in Pixels. All Border, Spacing, Padding, and Preferred Child Widths Are Added

Notes	Pixels
Left side <code>HBox</code> 's border stroke width	1
Left side padding inset	1
Left side of rectangle 1's margin width	2
Width of rectangle 1	10
Right side of rectangle 1's margin width	2
Spacing between Rectangle 1 and 2	5
Width of rectangle 2	20
Spacing between Rectangle 2 and 3	5
Width of rectangle 3	5
Spacing between rectangle 3 and 4	5
Width of rectangle 4	20
Right side of <code>HBox</code> 's padding inset	1
Right side of <code>HBox</code> 's border stroke width	1
Total width in pixels:	78

The total height in pixels of the HBox instance consists of the following in Table 5-2.

Table 5-2. The HBox's Total Height in Pixels Is Represented by Adding the Border, Spacing, Padding, and Preferred Child Widths

Notes	Pixels
Top side HBox's border stroke width	1
Top padding inset HBox between border and rectangle	1
Preferred Height of the largest rectangle (r2)	20
Padding inset of the HBox (bottom)	1
Border stroke width (bottom)	1
Total height of HBox in pixels:	24

Note When using a parent that is resizable, such as a Border pane with the HBox instance in the center, the HBox node's width and height will stretch. In this scenario, with non-resizable children, the only parts that get stretched are the border and padding around the HBox, which is the inset of the BorderPane parent. The dashed border essentially looks like it is surrounding the border pane center region taking up the available width and height.

VBox

The [VBox](#) layout is similar to an HBox, except that it places child nodes stacked in a vertical column. As children are added, each is placed beneath the previous child node. By default, the VBox honors the children's preferred width and height. When the parent node is not resizable (for example, a Group node), the maximum column's width is based on the child node with the greatest preferred width. The same goes for the VBox's overall height, which is based on the border, padding, spacing, margins, and child node's height all added up.

Also, by default, each child node aligns to the top-left (`Pos.TOP_LEFT`) position. To change the default alignment, use the `setAlignment(Pos value)` method. Refer to the Javadoc documentation at <https://docs.oracle.com/javase/8/javafx/api/javafx/geometry/Pos.html> to see all possible positions to align child nodes within a cell. VBox layouts can be configured to have resizing ranges to allow the parent node to manage and calculate space available upon layout requests. Remember, to resize the VBox and its children, the VBox's parent must also be resizable such as the BorderPane instance's center region.

A VBox Example

To demonstrate the use of a VBox, the example in Listing 5-2 contains the same four rectangles from the previous example, with the same layout constraints. In the same manner as the HBox example, the code will use a one-pixel stroke width to draw a dashed border (colored in blue) with an inset padding of one pixel.

Listing 5-2. The VBoxExample.java File Demonstrating VBox Layout

```
VBox vbox = new VBox(5);           // spacing between child nodes only.
vbox.setPadding(new Insets(1)); // space between vbox border and child nodes column

// The border is blue, dashed, 0% radius for all corners,
// a width of 1 pixel
BorderStroke [] borderStrokes = new BorderStroke[] {
    new BorderStroke(Color.BLUE,
                      BorderStrokeStyle.DASHED,
                      new CornerRadii(0.0, true),
                      new BorderWidths(1.0))
};

vbox.setBorder(new Border(borderStrokes));

Rectangle r1 = new Rectangle(10, 10); // little square
Rectangle r2 = new Rectangle(20, 20); // big square
Rectangle r3 = new Rectangle(5, 20); // vertical rectangle
Rectangle r4 = new Rectangle(20, 5); // horizontal rectangle

VBox.setMargin(r1, new Insets(2,2,2,2)); // margin around r1

vbox.getChildren().addAll(r1, r2, r3, r4);

root.getChildren().add(vbox);
primaryStage.setOnShown((WindowEvent we) -> {
    System.out.println("vbox width " + vbox.getBoundsInParent().getWidth());
    System.out.println("vbox height " + vbox.getBoundsInParent().getHeight());
});
```

Figure 5-3 shows the output of the `VBoxExample.java` application from Listing 5-2, demonstrating the `VBox` layout with rectangle shapes as child nodes. The `VBox` has a one-pixel width dashed border and a one-pixel padding inset. The one-pixel padding inset is between the border and the rectangles. The `VBox`, like a vertical bookshelf each shelf represents the spacing between the rectangles (horizontal white area). The spacing between the rectangles is defined by the `VBox`'s constructor or by using the `setSpacing()` method.

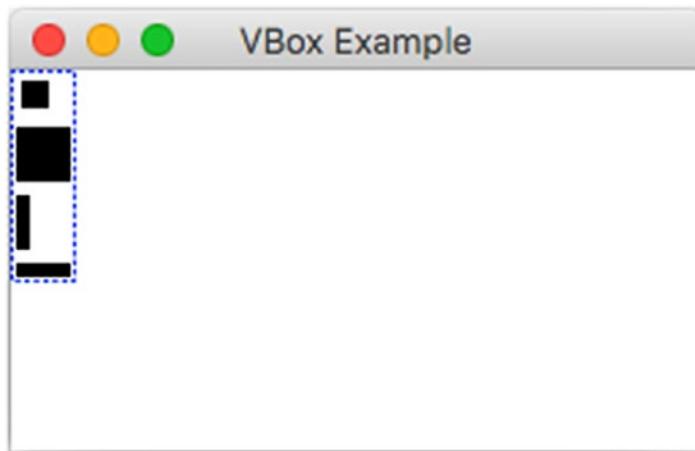


Figure 5-3. The output of `VBoxExample.java` depicting four rectangles added to a `VBox`

Code Walkthrough

In Listing 5-2, the code begins by instantiating a `VBox` with a five-pixel wide space between child nodes. Figure 5-4 depicts horizontal gray bars that represent the spacing between child nodes (`r1` to `r4`). After constructing the `VBox`, the code sets the padding via the `setPadding()` method. Setting the padding by using an `Insets` object allows you to create padding between the border and the row. In Figure 5-4, the `VBox` has a one-pixel wide dashed border with a padding of one pixel surrounding the child nodes.

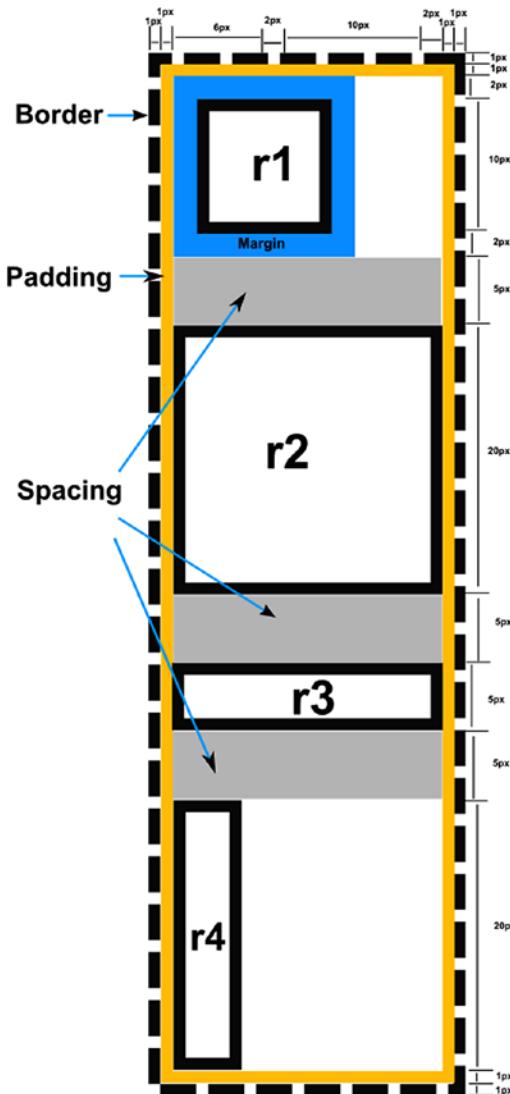


Figure 5-4. `VBox` having a one-pixel dashed border, a one-pixel padding, and four rectangle shaped nodes separated by five-pixel spacing

Once the four rectangles are created, the code sets the margin on the rectangle `r1`. To set the margin, we use the static method `VBox.setMargin(Node, Insets)`. The code then creates an `Insets` object with a margin of two pixels for the top, right, bottom, and left of rectangle 1 (`r1`). Rectangle 1 (`r1`) has a two-pixel margin shown in blue in Figure 5-4. To find the total size of a `VBox` instance, you simply add the spacing, margin, padding, and child nodes' dimensions (local bounds). Having a `Group` as a parent, the `VBox` node's calculated width would total to 24 pixels, and the height would total to 78 pixels.

Table 5-3. The `VBox`'s Total Width in Pixels. All Border, Spacing, Padding, and Largest Preferred Child Widths Are Added

Notes	Pixels
Left side <code>VBox</code> 's border stroke width	1
Left side padding inset	1
The width of the widest child node (Rectangle 2)	20
Right side padding inset between rectangle and border	1
Right side <code>VBox</code> 's border stroke width	1
Total width in pixels:	24

The total height in pixels of the `VBox` instance is shown in Table 5-4.

Table 5-4. The `VBox`'s Total Height in Pixels. All Border, Spacing, Padding, and Preferred Child Widths Are Added

Notes	Pixels
Top of <code>VBox</code> 's border stroke width	1
Padding inset (top) between border and child node	1
Rectangle1's margin (top) width	2
Width of rectangle 1	10
Rectangle 1's margin (bottom) width	2
Spacing between Rectangle 1 and 2	5
Height of rectangle 2	20
Spacing between Rectangle 2 and 3	5
Height of rectangle 3	5
Spacing between rectangle 3 and 4	5
Height of rectangle 4	20
<code>VBox</code> 's padding inset between child node and border	1
<code>VBox</code> 's border stroke (bottom) width	1
Total width in pixels:	78

FlowPane

The FlowPane layout node allows child nodes in a row to flow based on the available horizontal spacing and wraps nodes to the next line when horizontal space is less than the total of all the nodes' widths. Figure 5-5 shows four child nodes flowing from left to right and then wrapping.

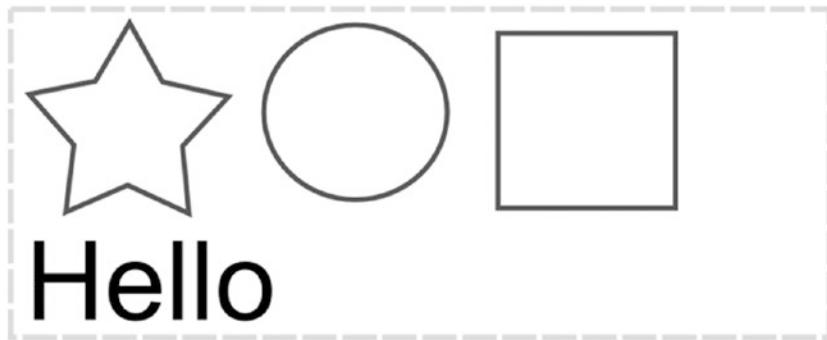


Figure 5-5. A FlowPane layout with nodes flowing left to right with the ability to wrap nodes based on horizontal space. The Hello text node is wrapped to the next line because the available horizontal space, denoted by the dashed line, is too small.

By default, a FlowPane layout flows child nodes from left to right (`Pos.TOP_LEFT`). To change the flow alignment, you can simply invoke the `setAlignment()` method by passing in an enumerated value of type `Pos`. Listing 5-3 creates a FlowPane layout to flow child nodes from right to left (`Pos.TOP_RIGHT`).

Listing 5-3. The Creation of a FlowPane Layout to Flow from Right to Left

```
FlowPane flowPane = new FlowPane();
flowPane.setAlignment(Pos.TOP_RIGHT);
flowPane.getChildren().addAll(...); // child nodes to add.
```

BorderPane

The BorderPane layout node allows child nodes to be placed in a top, bottom, left, right, or center regions. Because each region can only have one node, developers typically nest layouts. An example would be to create an HBox with children to then be set as the top region via the `setTop()` method. Figure 5-6 shows the BorderPane's regions.

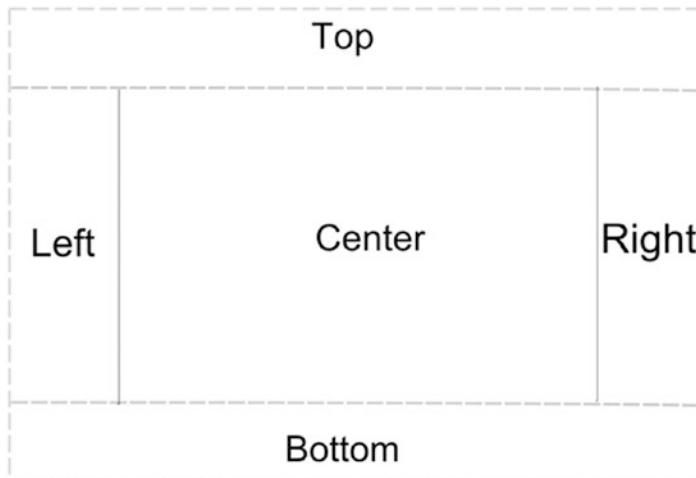


Figure 5-6. The `BorderPane` layout

This layout is similar to many web sites you see every day, where navigational links are placed in the top, bottom, left, or right regions of a page, while the main content is in the center area. A `BorderPane`'s region of top and bottom borders allows a resizable node to take up all of the available width. The left and right border regions take up the available vertical space between the top and bottom borders. All of the bordering regions honor the children's sizes (preferred width and height) by default. According to the Javadocs, the default alignment of nodes, when placed in the top, bottom, left, right, and center regions, is as follows:

- Top: `Pos.TOP_LEFT`
- Bottom: `Pos.BOTTOM_LEFT`
- Left: `Pos.TOP_LEFT`
- Right: `Pos.TOP_RIGHT`
- Center: `Pos.CENTER`

Next, you will be learning about the `GridPane` layout in combination with the `BorderPane` to create a simple UI similar to a `Contacts` form-like application. By using the `BorderPane` and its center region as the root node, the available horizontal and vertical space will be given to the child nodes. This will allow text fields to stretch when the window is being resized.

GridPane

So far you have learned about simple layouts; now let's take a look at a more advanced layout, `GridPane`. It's commonly used in business applications. For instance, business applications typically have data entry form screens. Forms usually have read-only labels on the first column and input fields on the second column, resembling a grid pattern.

When using the `GridPane` class, you can specify constraints at the row, column, or cell level. For example, if the second column contains input text fields, you may want them to resize as the window is being resized (width-wise). To lay out components in a grid-like manner, I created an example form-type application in Listing 5-4 that uses JavaFX's `GridPane` layout. The most interesting part of the example is the use of column constraints on the `GridPane` layout node. This is convenient because any nodes placed in a column would take on the column constraint instead of having to configure each node individually. Column constraints apply a minimum and maximum horizontal width for the child nodes to grow or shrink as the window is resized.

A Form-Type Application Example

To demonstrate a `GridPane` layout, Listing 5-4 is a simple form type UI that resembles a contacts application allowing users to enter a first and last name followed by a Save button. To make things a little more professional, the code creates a banner header containing a logo (SVG icon).

Listing 5-4. The `GridPaneForm.java` File Demonstrates the `GridPane` Layout

```
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("GridPaneForm ");
    BorderPane root = new BorderPane();
    Scene scene = new Scene(root, 380, 150, Color.WHITE);

    GridPane gridpane = new GridPane();
    //gridpane.setGridLinesVisible(true);
    gridpane.setPadding(new Insets(5));
    gridpane.setHgap(5);
    gridpane.setVgap(5);
    ColumnConstraints column1 = new ColumnConstraints(100);
    ColumnConstraints column2 = new ColumnConstraints(50, 150, 300);
    column2.setHgrow(Priority.ALWAYS);
    gridpane.getColumnConstraints().addAll(column1, column2);

    Label fNameLbl = new Label("First Name");
    TextField fNameFld = new TextField();
    Label lNameLbl = new Label("Last Name");
    TextField lNameFld = new TextField();

    Button saveButt = new Button("Save");

    // First name label
    GridPane.setAlignment(fNameLbl, HPos.RIGHT);
    gridpane.add(fNameLbl, 0, 0);

    // Last name label
    GridPane.setAlignment(lNameLbl, HPos.RIGHT);
    gridpane.add(lNameLbl, 0, 1);

    // First name field
    GridPane.setAlignment(fNameFld, HPos.LEFT);
    gridpane.add(fNameFld, 1, 0);

    // Last name field
    GridPane.setAlignment(lNameFld, HPos.LEFT);
    gridpane.add(lNameFld, 1, 1);

    // Save button
    GridPane.setAlignment(saveButt, HPos.RIGHT);
    gridpane.add(saveButt, 1, 2);
```

```

// Build top banner area
FlowPane topBanner = new FlowPane();
topBanner.setAlignment(Pos.TOP_LEFT);
topBanner.setPrefHeight(40);
String backgroundStyle =
        "-fx-background-color: lightblue;" +
        "-fx-background-radius: 3px;" +
        "-fx-background-inset: 5px;";
topBanner.setStyle(backgroundStyle);
SVGPath svgIcon = new SVGPath();
// icon from http://raphaeljs.com/icons/
svgIcon.setContent("M21.066,20.667c1.227-0.682,1.068-3.311-0.354-5.874c-0.611-
1.104-1.359-1.998-2.109-2.623c-0.875,0.641-1.941,1.031-3.102,1.031c-1.164,
0-2.231-0.391-3.104-1.031c-0.75,0.625-1.498,1.519-2.111,2.623c-1.422,2.563-1.578,
5.192-0.35,5.874c0.549,0.312,1.127,0.078,1.723-0.496c-0.105,
0.582-0.166,1.213-0.166,1.873c0,2.938,1.139,5.312,2.543,5.312c0.846,0,1.265-0.865,
1.466-2.188c0.2,1.314,0.62,2.188,1.461,2.188c1.396,0,2.545-2.375,2.545-5.312c0-0.66-
0.062-1.291-0.168-1.873C19.939,20.745,20.516,20.983,21.066,20.667zM15.5,
12.201c2.361,0,4.277-1.916,4.277-4.279S17.861,3.644,15.5,3.644c-2.363,
0-4.28,1.916-4.28,4.279S13.137,12.201,15.5,12.201zM24.094,14.914c1.938,0,3.512-
1.573,3.512-3.513c0-1.939-1.573-3.513-3.512-3.513c-1.94,0-3.513,1.573-3.513,3.513C20.
581,13.341,22.153,14.914,24.094,14.914zM28.374,17.043c-0.502-0.907-1.116-1.641-1.732-
2.154c-0.718,0.526-1.594,0.846-2.546,0.846c-0.756,0-1.459-0.207-2.076-0.55c0.496,1.09
3,0.803,2.2,0.861,3.19c0.093,1.516-0.381,2.641-1.329,3.165c-0.204,0.117-0.426,0.183-
0.653,0.224c-0.056,0.392-0.095,0.801-0.095,1.231c0,2.412,0.935,4.361,2.088,4.36
1c0.694,0,1.039-0.71,1.204-1.796c0.163,1.079,0.508,1.796,1.199,1.796c1.146,0,
2.09-1.95,2.09-4.361c0-0.542-0.052-1.06-0.139-1.538c0.492,0.472,0.966,0.667,
1.418,0.407C29.671,21.305,29.541,19.146,28.374,17.043zM6.906,14.914c1.939,0,
3.512-1.573,3.512-3.513c0-1.939-1.573-3.513-3.512-3.513c-1.94,
0-3.514,1.573-3.514,3.513C3.392,13.341,4.966,14.914,6.906,14.914zM9.441,21.536c-1.593-
0.885-1.739-3.524-0.457-6.354c-0.619,0.346-1.322,0.553-2.078,0.553c-0.956,
0-1.832-0.321-2.549-0.846c-0.616,0.513-1.229,1.247-1.733,2.154c-1.167,2.104-
1.295,4.262-0.287,4.821c0.451,0.257,0.925,0.064,1.414-0.407c-0.086,0.479-
0.136,0.996-0.136,1.538c0,2.412,0.935,4.361,2.088,4.361c0.694,0,1.039-0.71,1.204-
1.796c0.165,1.079,0.509,1.796,1.201,1.796c1.146,0,2.089-1.95,2.089-4.361c0-0.432-0.04-
0.841-0.097-1.233C9.874,21.721,9.651,21.656,9.441,21.536z");
svgIcon.setStroke(Color.LIGHTGRAY);
svgIcon.setFill(Color.WHITE);

Text contactText = new Text("Contacts");
contactText.setFill(Color.WHITE);

Font serif = Font.font("Dialog", 30);
contactText.setFont(serif);
topBanner.getChildren().addAll(svgIcon, contactText);

root.setTop(topBanner);
root.setCenter(gridpane);

primaryStage.setScene(scene);

primaryStage.show();
}

```

Code Walkthrough

After executing Listing 5-4, you will see the output in Figure 5-7, which displays a simple form application allowing the users to enter a first and last name as contact information. When resizing the window, notice that the input text field controls grow or shrink based on the column constraints. Also, you'll notice that the Save button is horizontally aligned to the right (`HPos.RIGHT`), just beneath the Last Name text field.



Figure 5-7. Output of the `GridPaneForm.java` file

Listing 5-4 begins by creating an instance of a `BorderPane` object as the root node for the scene. By using a `BorderPane` as the root node and then placing the `GridPane` node object into the center region, you enable the parent (`BorderPane`) to resize the `GridPane` by giving it all the available horizontal and vertical space. After creating a `BorderPane` instance as the root node, the code instantiates a `GridPane` layout with padding along with horizontal and vertical gaps. Listing 5-5 then creates a `GridPane` layout with padding, horizontal, and vertical gaps set to 5 (pixels).

Listing 5-5. Setting Padding, Horizontal, and Vertical Gap Spacing Between UI Controls for Each Cell

```
GridPane gridpane = new GridPane();
gridpane.setPadding(new Insets(5));
gridpane.setHgap(5); gridpane.setVgap(5);
```

Before going further, I want to point out a handy method on the `GridPane` object called `setGridLinesVisible()`. This allows you to debug your grid layout's cell constraints. After invoking the `setGridLinesVisible(true)` method, gridlines will appear surrounding its cells and gaps.

■ Tip In Figure 5-8, the grid pane's `gridlinesVisible` property set to `true` can help you debug cell constraints. To turn on the gridlines, invoke the method `setGridLinesVisible()` on your grid pane instance.

Figure 5-8 shows the grid pane with gridlines visible.



Figure 5-8. The grid pane's `gridlines visible` property set to true. The `visible` property is often used for debugging purposes. To turn on the gridlines, invoke the method `setGridLinesVisible(true)`.

Continuing on with the code walkthrough in Listing 5-4, the code implements column constraints for the first and second columns. To set a column's constraint, use the `ColumnConstraints` constructor, which allows the developer to specify minimum, preferred, and maximum widths for a column on the `GridPane` node. This is very nice; it means that when the parent container node (`BorderPane`) is being resized, the `GridPane` will cause its child nodes to grow or shrink horizontally based on its row or column constraints. Listing 5-6 demonstrates the ability to constrain the first column to 100 pixels wide for the read-only labels and the second column with a minimum width of 50, a preferred width of 150 and a maximum width of 300.

Listing 5-6. Setting Up Column Constraints for a JavaFX Application

```
ColumnConstraints column1 = new ColumnConstraints(100); // fixed for labels
ColumnConstraints column2 = new ColumnConstraints(50, 150, 300); // min,pref,max
column2.setHgrow(Priority.ALWAYS);
gridpane.getColumnConstraints().addAll(column1, column2);
```

After creating the column constraint for column 2, the code invokes the `setHgrow()` method by passing in the enum value `Priority.ALWAYS`, which lets the UI controls take up the available horizontal space within cells when the window is widened. I set the maximum width to 300 for the second column so you can see that the child `TextField` does not grow beyond the maximum width (300). In other words, when the window is resized wider than 300 pixels, the text fields within the grid pane column stop stretching.

The next step is simply putting each UI control into its respective cell location. All cells are relative to zero; to place a node in the cell in the first column and first row, you would specify the cell as (0, 0). Thus, the following code snippet adds the Save button into the second column, third row of the `GridPane` layout at cell (1, 2):

```
gridpane.add(saveButton, 1, 2);
```

The layout also allows you to align controls horizontally or vertically within the cell. The following statement right-aligns the Save button:

```
GridPane.setAlignment(saveButton, HPos.RIGHT);
```

After completing the `GridPane` setup, the code implements the top banner area of the input form shown in Figure 5-7 earlier. To create a title banner, I used a flow layout with a background color of light blue, an SVG node icon (shown as people), and a text node with the text “Contacts”. I obtained the SVG based icon from Dmitry Baranovskiy, founder and author of the JavaScript library RaphaelJS. The web site that hosted the original SVG icons is no longer active. The icon is in the form of a string in SVG notion (vector drawing commands). I simply pass the SVG string to an `javafx.scene.shape.SVGPath` instance by invoking the `setContent()` method. The following snippet shows how to create an `SVGPath` node instance.

```
SVGPath svgIcon = new SVGPath();
svgIcon.setContent("M21.066,...");
```

The important code that relates to the `FlowPane` layout displaying the banner contains the settings for the preferred height and JavaFX CSS styles. Because the banner (`FlowPane`) is placed in the top region of the `BorderPane` layout, the preferred height of 40 pixels will be honored and the width will take up the available horizontal space as the window is being resized (stretched wider).

With this knowledge that the `FlowPane` takes up the available horizontal space, you’ll notice the area being filled with the light blue background color. By using JavaFX CSS styling, the code sets a color of light blue along with corner radii and insets specified using a string (`backgroundStyle`). Listing 5-7 is the code section that styles the banner (`FlowPane`) that is placed in the top region of the `BorderPane` layout. Don’t worry if JavaFX CSS styling doesn’t make sense yet; a more detailed discussion of it will follow in Chapter 15 on Custom UIs.

Listing 5-7. The styling code for a `FlowPane` layout occupying the top banner area having a light blue color as the background

```
FlowPane topBanner = new FlowPane();
topBanner.setPrefHeight(40);
String backgroundStyle = "-fx-background-color: lightblue;" +
    "-fx-background-radius: 30%;"
    "-fx-background-inset: 5px;";
topBanner.setStyle(backgroundStyle);
```

We’ve only scratched the surface of the `GridPane` layout. There are other ways to provide constraints and alignment of child nodes, so I trust you will delve deeper into the `GridPane` layout’s APIs. See the Javadocs documentation for more details. Next, you are going to learn about creating UIs using a graphical editor called Scene Builder.

Scene Builder

Up to this point, you’ve learned how to code and style UI elements programmatically. You are probably wondering if there is a better way to build UIs. Well, I am happy to inform you that the JavaFX team at Oracle built a free tool called Scene Builder, which allows you to build UIs graphically, often called WYSIWYG editors. (WYSIWYG stands for What You See Is What You Get.) After some time, Oracle announced that they would no longer provide downloadable binaries of the Scene Builder tool.

Since the Scene Builder tool has been open sourced, the task of building and hosting downloadable binaries has since been immediately picked up by the company GluonHQ (see <http://gluonhq.com/>). In this and many other bold moves, GluonHQ forged ahead with other products and services that would empower Java developers. GluonHQ’s main offering is to provide a UI and cloud development platform for companies to develop apps with a single codebase (Java) and the capability to deploy onto many platforms such as mobile, desktop, and embedded devices. Typical platforms are iOS, Android, Windows, MacOS X, and Raspbian (Raspberry Pi).

In this section, you will learn how to build a simple UI and use the serialized UI formatted file (FXML) in a JavaFX application. To get started, head over to GluonHQ (<http://gluonhq.com>) to download the Scene Builder tool at the following location:

<http://gluonhq.com/open-source/scene-builder/>

Download and Installing Scene Builder

As of the writing of this book, a version of the Scene Builder tool for JDK 9 didn't exist. However, you can simply point your `JAVA_HOME` environment variable to the latest Java 8 SDK to run a downloaded Scene Builder version 8. Probably by the time you read this there will already be a version available for Java 9. Assuming you've already downloaded the Scene Builder tool for your appropriate OS, to launch the tool, I recommend downloading the JAR version and launching it using the command prompt. In my opinion, using the command prompt is better because it will assist you by dumping out any errors and exceptions to the console. The following command will launch the Scene Builder tool from the command line.

```
$ java -jar SceneBuilder-8.0.0.jar
```

Launching Scene Builder

After downloading and installing Scene Builder, you can launch it. Figure 5-9 shows the Scene Builder tool.

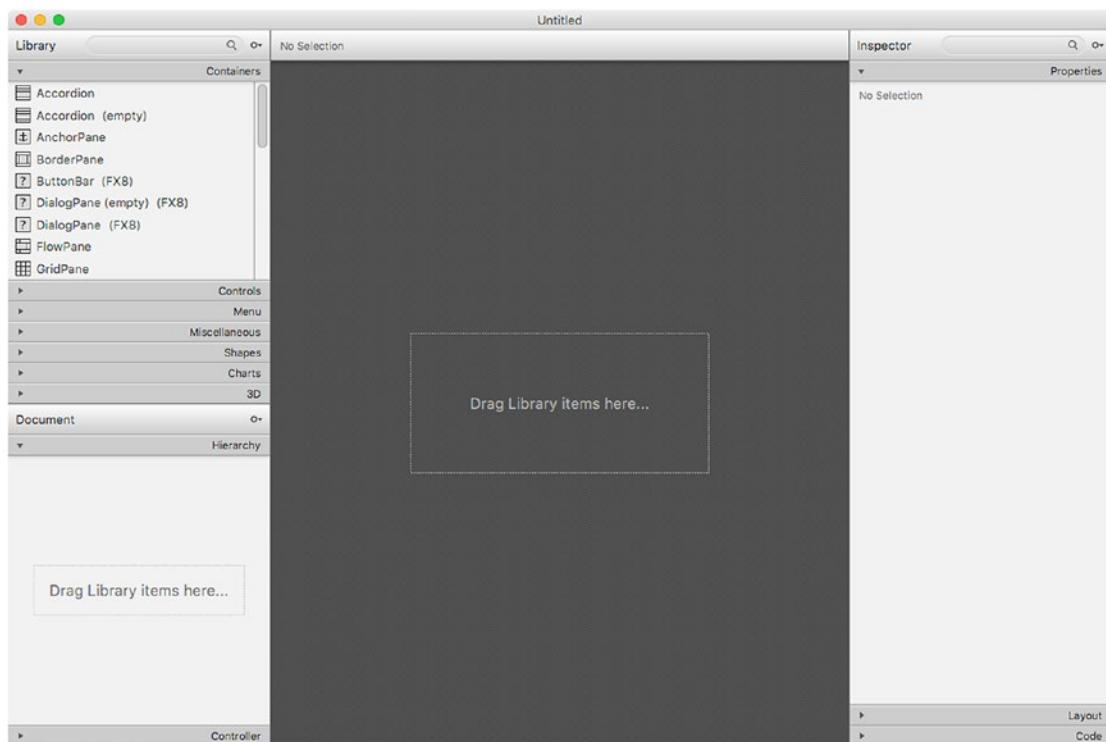


Figure 5-9. Launching the Scene Builder tool for the first time

I begin by describing the various tabs and their subsections starting with the left side. Under the main tab titled Library, note the search bar. There are also subsections called Containers, Controls, Menu, Miscellaneous, Shapes, Charts, and 3D. These subsections are various kinds of JavaFX nodes that can be dragged onto the center canvas area (*Drag Library items here...*). Of course, the first type of JavaFX node must be a layout container type node, which will allow children type nodes such as controls to be dragged onto container type nodes.

On the left beneath the UI elements is the Document section. The Document section has two subsections, called Hierarchy and Controller. The Hierarchy panel allows you to view the DOM (document object model) of the UI elements in a tree-like hierarchy. Lastly, below the Hierarchy is the Controller tab. The Controller section lets you reference control logic (Controller Java class) to this UI view (FXML). The Controller class that is referenced will allow handler code to interact with UI elements that were previously dragged onto the canvas area. An example would be a Save button that is clicked would invoke a method on an associated Controller class. Later, you will be creating a Controller class to be referenced in the FXML file.

On the right side of the Scene Builder is the Inspector search bar, as shown in Figure 5-9. The search bar allows you to find properties on UI elements quite easily. Underneath the Inspector search bar, you'll see three subsections—Properties, Layout, and Code. The Properties panel section allows you to style UI elements using JavaFX CSS properties. In Chapter 15 you will learn more about CSS styling, but for now you will not be using JavaFX CSS styling on UI elements in the tool. The Layout panel allows you to specify dimensions, transforms, and padding of a node's properties. Lastly is the Code panel, which allows you to map event methods to UI elements such as the OnMousePressed property.

Now that you know what Scene Builder can do, let's build a user interface (UI). But, before building a UI, you will want to know what kind of UI you are wanting to build. To keep things simple, you will build the familiar Contacts form you created earlier in the section on GridLayout. Instead of programmatically coding the UI, you are going to develop it using the Scene Builder tool. Follow the steps below to graphically build your form.

1. Drag a BorderPane container onto the canvas to the right. Figure 5-10 shows dragging the BorderPane container onto the canvas area.

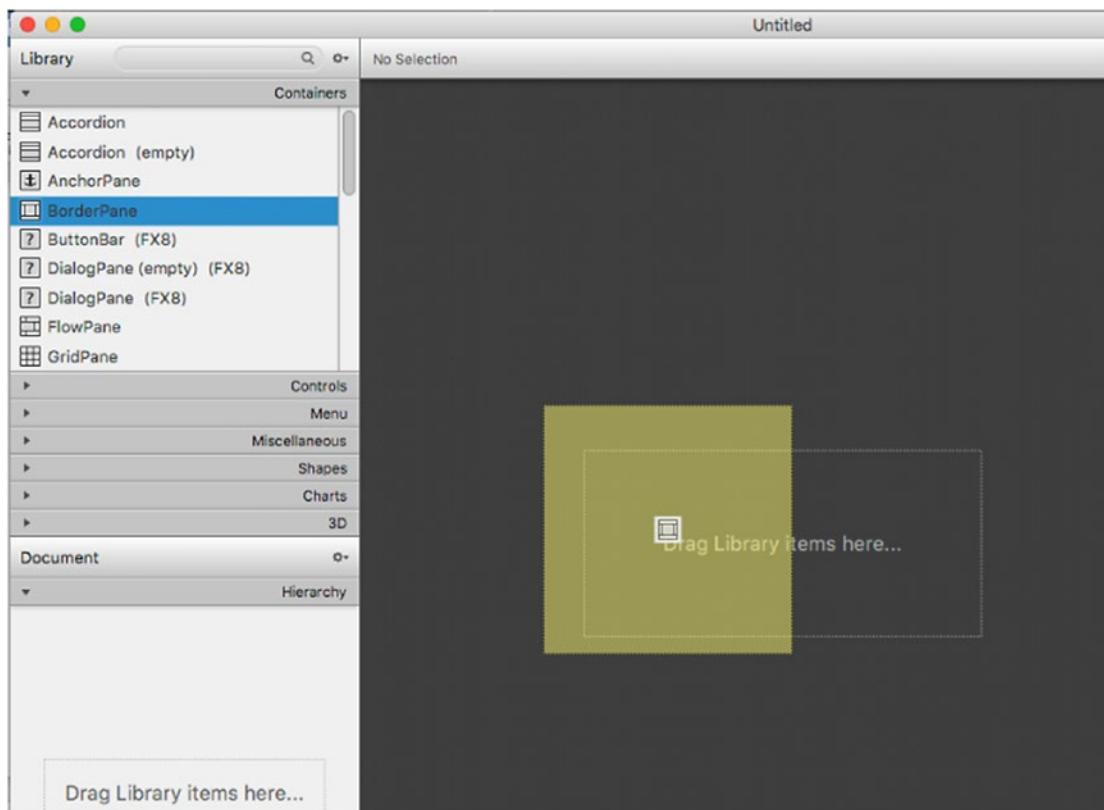


Figure 5-10. Dragging a BorderPane onto the canvas area that reads “Drag Library items here...”

After dragging the BorderPane container onto the canvas surface, you should see in the lower-left the Document tab under the Hierarchy section displaying the BorderPane's regions—TOP, LEFT, CENTER, RIGHT, and BOTTOM. Figure 5-11 shows the document hierarchy (DOM) of the current UI elements on the canvas area. You can also drag UI elements into the document hierarchy window. Sometimes it's easier to drag UI elements into the hierarchy and sometimes it's easier to drag UI elements onto the canvas area. It all depends on how small the regions are on the canvas when there are nested containers.

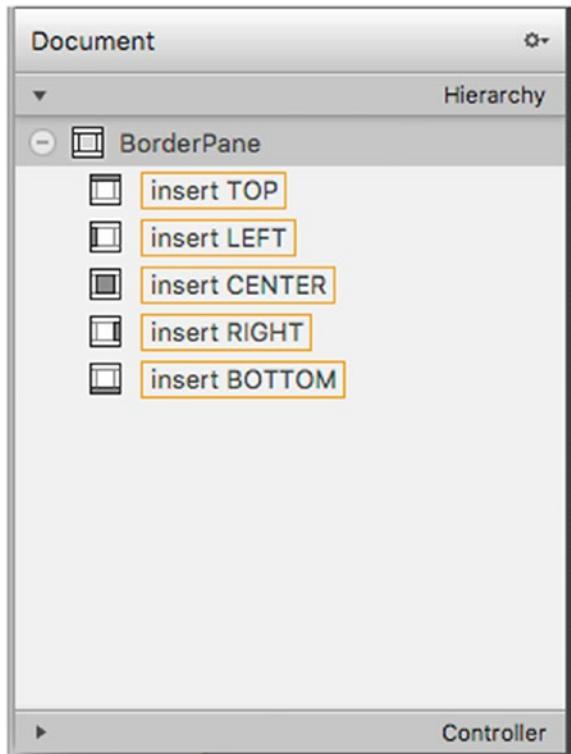


Figure 5-11. Document's hierarchy displaying the BorderPane's regions

2. Drag a `GridPane` container into the `BorderPane`'s center region, as shown in Figure 5-12. Figure 5-13 shows the `GridPane` node after the drag-and-drop operation onto the `BorderPane` node's center region. Notice the numbered columns and rows as zero relative cells allowing you to drag elements into them. Since the `BorderPane`'s center area takes up all the available width and height, when placing a `GridPane` onto the center region, the width and height of the `GridPane` will stretch.

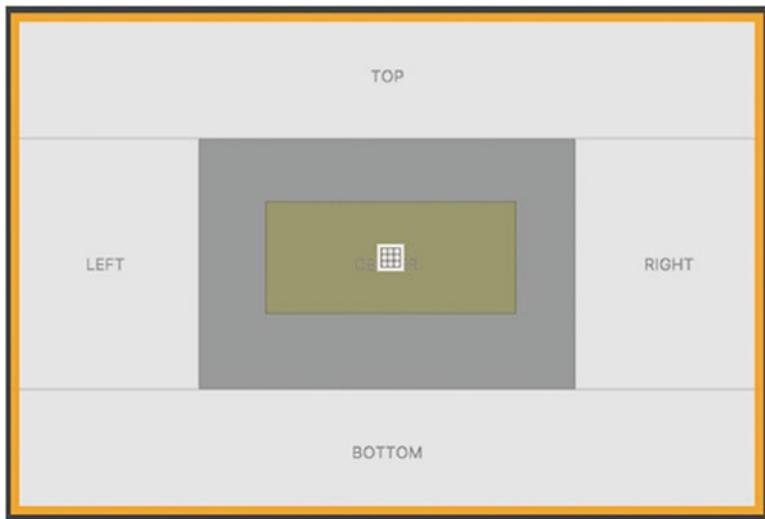


Figure 5-12. Dragging a `GridPane` node on to the center region of the `BorderPane` node

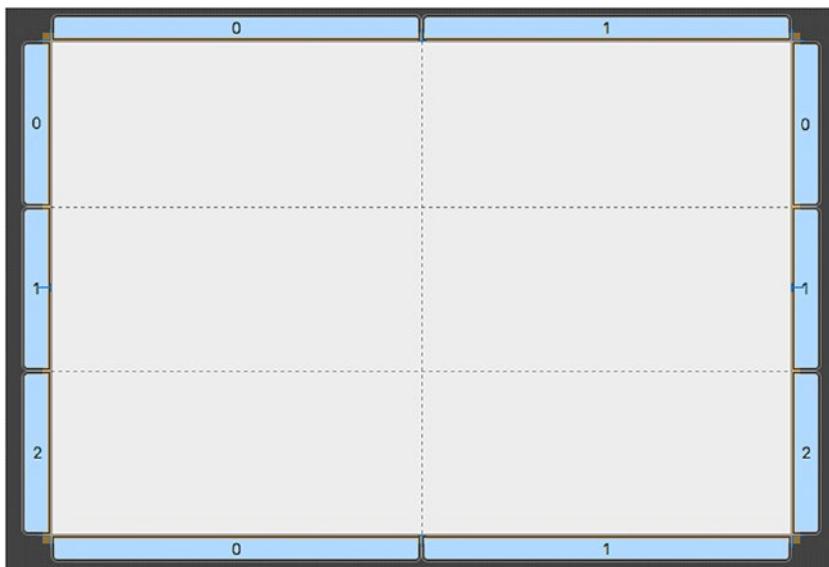


Figure 5-13. A `GridPane` positioned on the center region of a `BorderPane`

3. Before configuring the `GridPane`, you will want to reconfigure the `BorderPane`'s Preferred and maximum width to allow the `GridPane` to take up the available horizontal space. Select the `BorderPane` node by selecting it within the Hierarchy tab on the lower-left side beside the canvas area. Next, on the right side of the canvas area is the Layout tab (under Properties), you will be changing the Pref and Max Width properties of the `BorderPane`. Shown in Figure 5-14 are the preferred and maximum widths settings for the `BorderPane` node. Make sure the preferred width is `USE_COMPUTED_SIZE` and the maximum width is set to `MAX_VALUE`. This allows the `GridPane` node and its child nodes the opportunity to grow horizontally (stretch) when the JavaFX Stage window is being resized.

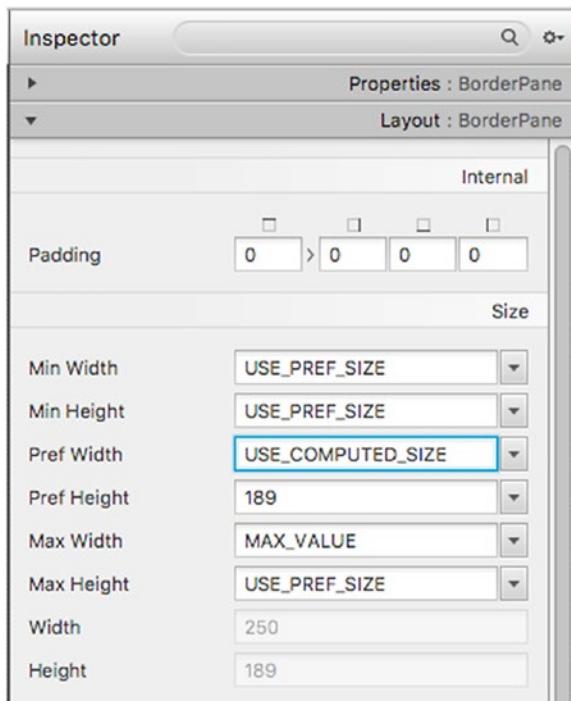


Figure 5-14. The `BorderPane`'s `Pref Width` and `Max Width` properties settings allow the `GridPane` to stretch horizontally

4. Set the GridPane layout's horizontal and vertical gap to five pixels and set the padding to five pixels. The padding and gap settings are on the right side of the canvas area, under the Layout tab section (see Figure 5-15). Next, you will configure the constraints for the first and second columns of the GridPane node. If you remember the first column will contain labels and the second column will contain text fields for our form.

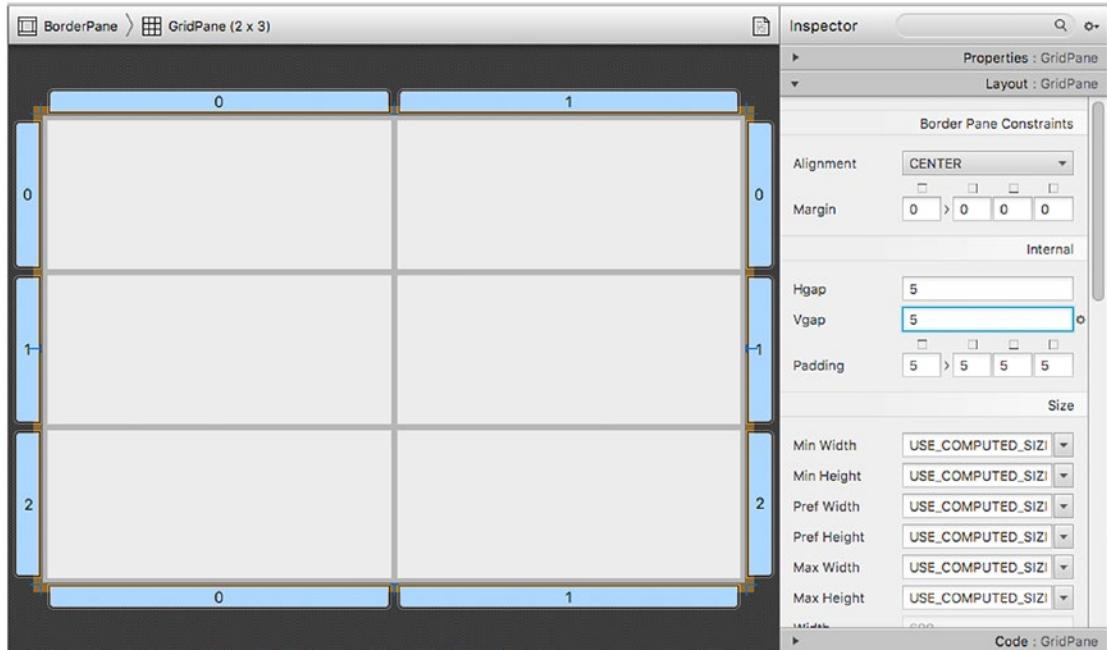


Figure 5-15. A GridPane layout's properties configured with five pixels for padding, Hgap, and Vgap

- Configure the GridPane's column constraints. To configure a column constraint, you will first select the column header zero (0) shown in Figure 5-16. You then configure the width under the Layout: ColumnConstraints property sheet. For the min and max widths, use the USE_PREF_SIZE option in the dropdown. In the pref width, enter 100. This means the column will not stretch because its minimum and maximum width will be set to the preferred width.

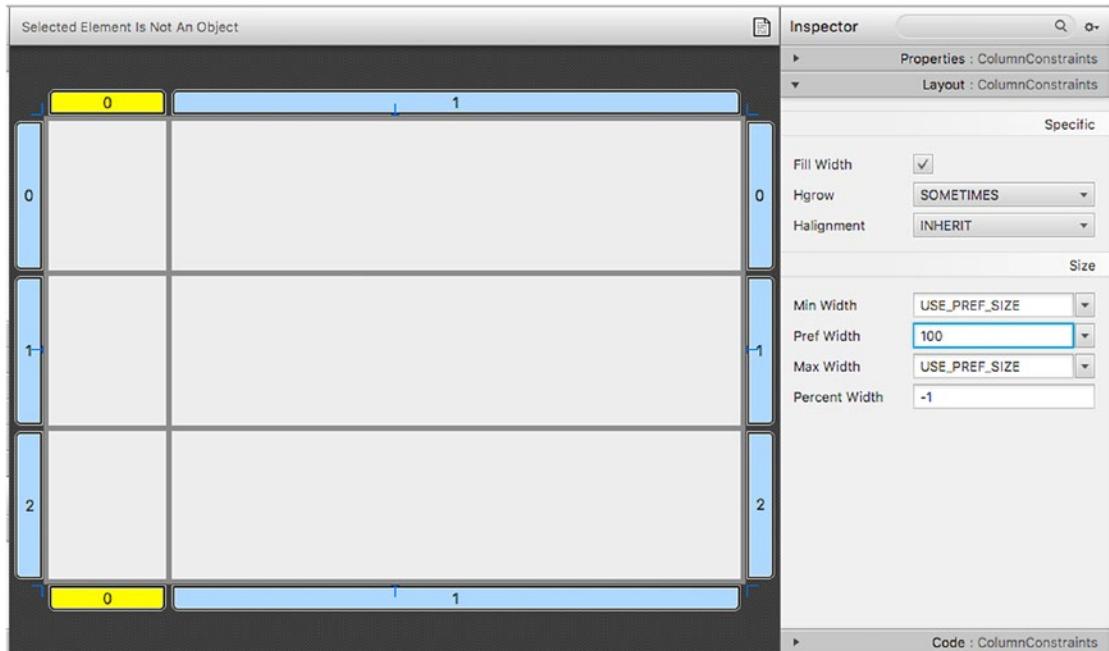


Figure 5-16. Setting the GridPane's first column (zero relative) constraint to a preferred width of 100

6. Configure the second GridPane's column constraints. To configure a column constraint, you select the column header one (1) shown in Figure 5-17. After selection, you configure the width under the Layout: ColumnConstraints property sheet.

Set the min, pref, and max width to 50, 150, and 300, respectively. You should also set the Hgrow property to always.

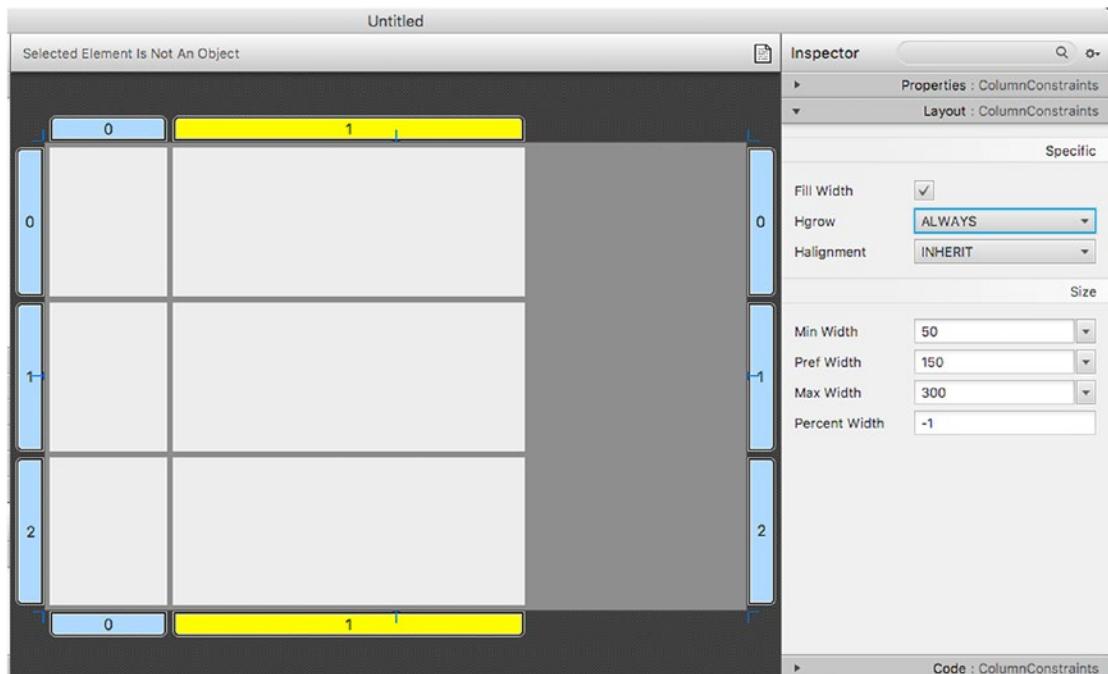


Figure 5-17. The GridPane's second column's constraint being configured

7. Configure the row constraints of the `GridPane`. Similar to selecting a column header, you will select a row header zero (left), as shown in Figure 5-18. Instead of selecting one row, hold down the Shift key and mouse click the row headers 0, 1, and 2, as shown in Figure 5-18. The objective is to set all three row constraints in one shot. You will be setting the three rows with a preferred height that will honor the UI element placed into it. For instance, when a label is placed into the row cell, the height of the label will be used as the height of the cell row. Select `USE_PREF_SIZE` for both min and max height properties. Enter 30 for the preferred height property.

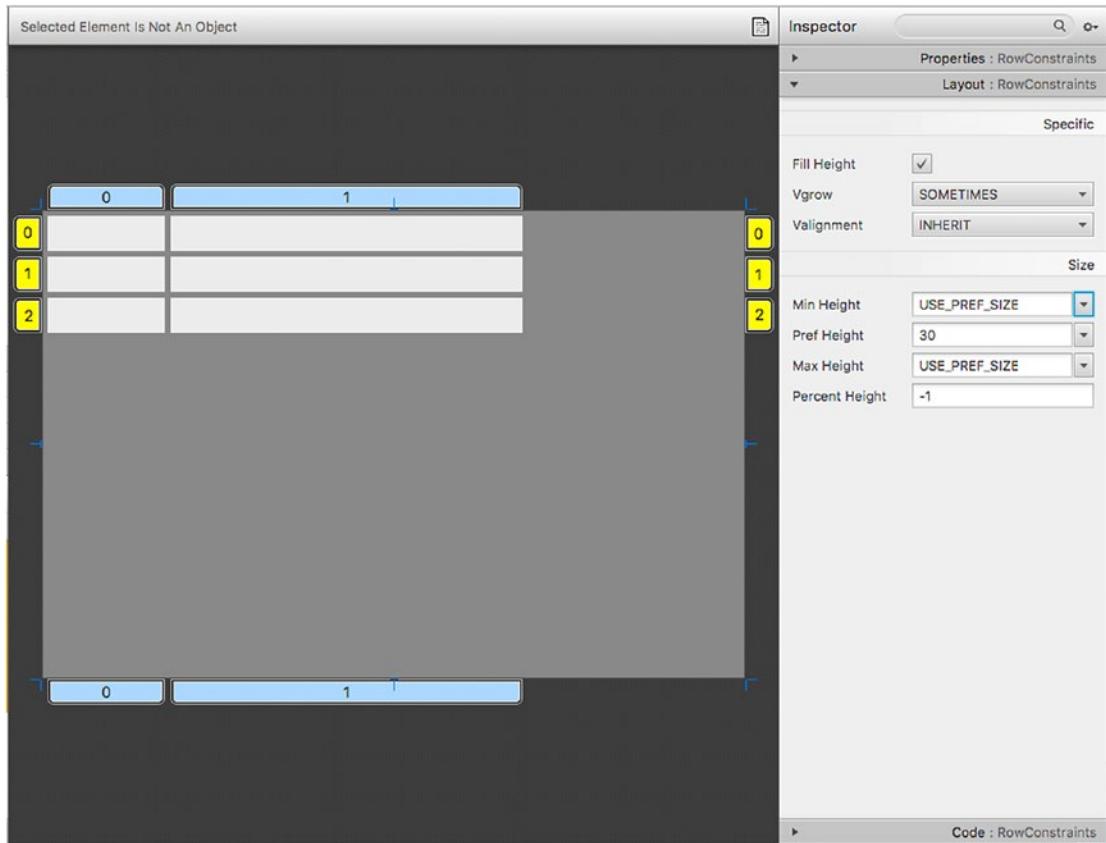


Figure 5-18. Setting the three rows of the `GridPane`'s row height constraints. The min and max height properties are set to `USE_PREF_SIZE`. The prefheight property is set to 30.

8. Create the first name label in the cell occupying the first column and first row. Drag a label from the Controls section under Library into the first column and the first row cell of the grid pane. This label will be the First Name. Figure 5-19 shows the text property set to First Name in the Properties tab subsection.

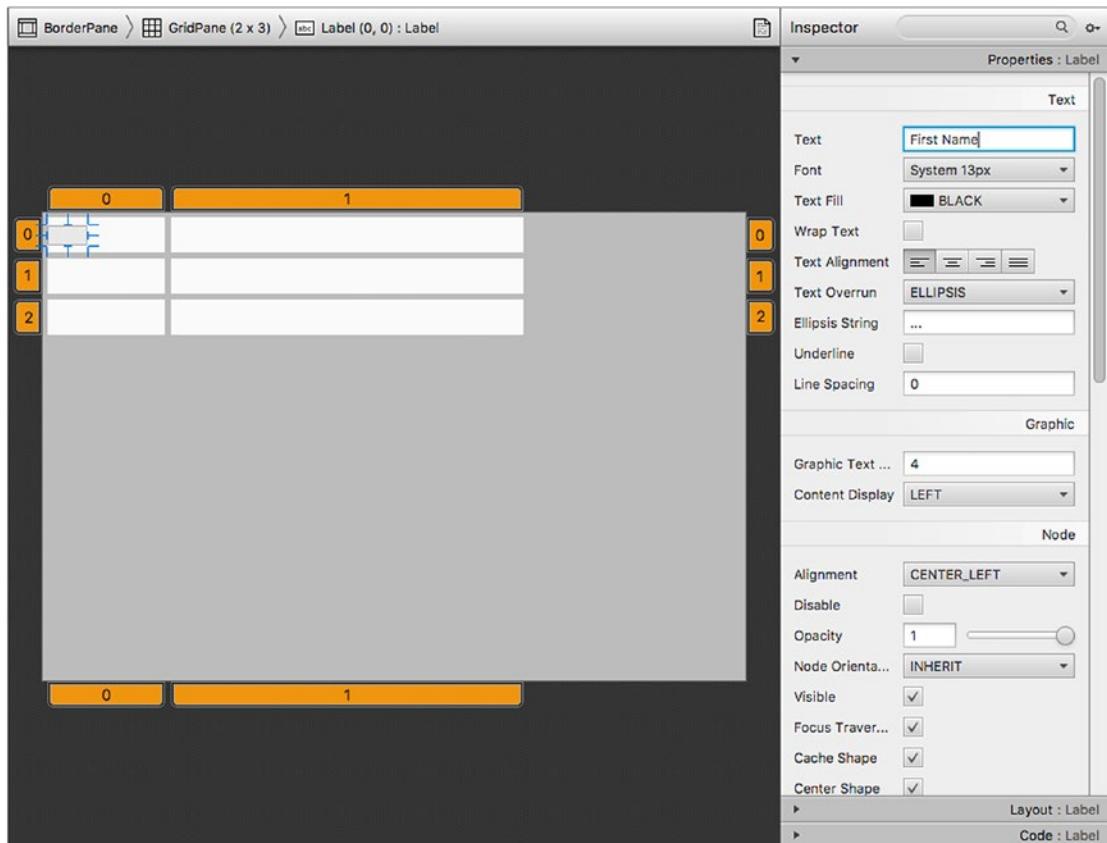


Figure 5-19. The first name label being added to the first column and first row cell. You'll notice under the Properties tab you will enter the text for the label.

- Set the label's horizontal alignment (Halignment) to be right-aligned. In Figure 5-20, the label's horizontal alignment is set by going to the Property sheet under the Layout tab subsection.

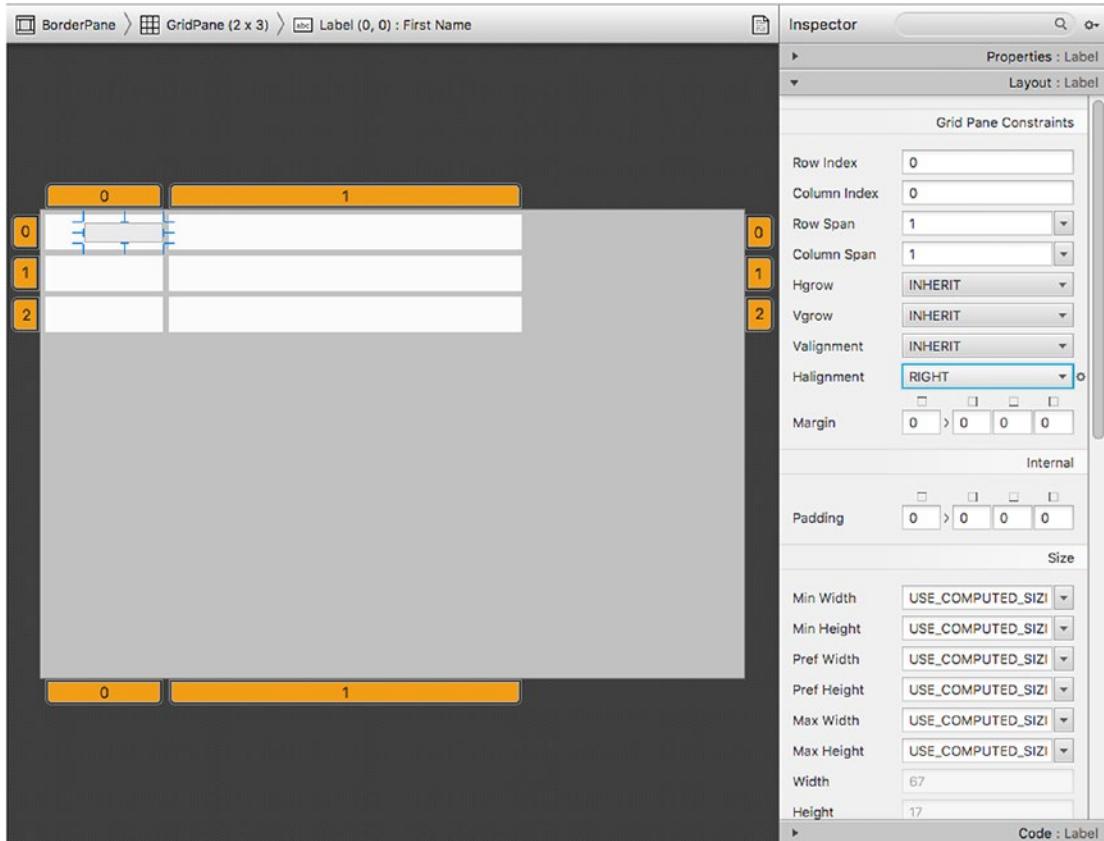


Figure 5-20. Setting a label in a GridPane cell's horizontal alignment to be right-aligned. Under the Layout: Label (Grid Pane Constraints), the Halignment is set to RIGHT.

10. Repeat Steps 8-9 to set up the Last Name label for the second row, first column. In the menu options, select Preview ► Show Preview in Window. Figure 5-21 shows the current UI elements previewed in a window.



Figure 5-21. A preview window showing the progress thus far—two labels, right horizontally aligned on the first column

11. Drag and drop the text field UI elements into the cell 1,0 and cell 1,1, which represent the first name and last name text fields, respectively. You'll also be dragging a JavaFX Button node to the second column, third row cell. When you preview your work, the UI should look like Figure 5-22.

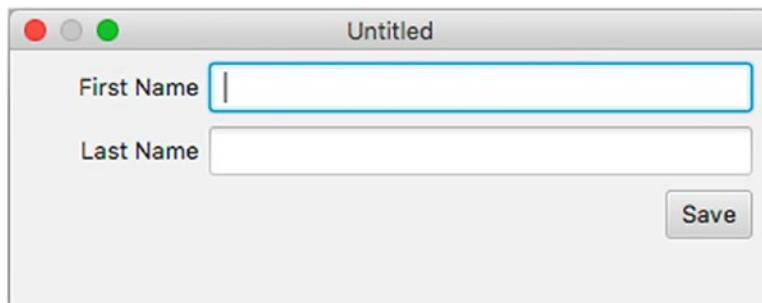


Figure 5-22. The preview window of the UI elements laid out onto the GridPane

12. Save the UI into an FXML file. In the File menu, select Save As and save it into a Java project's resource directory or into the src directory. Scene Builder allows you to save the UI into a file format called FXML having the file extension of *.fxml*. Save the file as *ContactForm.fxml*. Later, you will create code that will load the FXML file to be displayed and perform actions on UI elements. This file should be placed in a Java projects resources directory—such as `src/main/resources/ContactForm.fxml`. (You can create one folder to hold all three files for the project if you aren't using build tools such as Gradle or Maven. If you compile on the command line, make sure the FXML file is copied to the root classpath.) At this point, the current file (*ContactForm.fxml*) does not reference a Controller class or UI elements with the *fx:id* attributes. In a later step when you've created a Controller class, you will update the UI through Scene Builder and save the FXML file again.

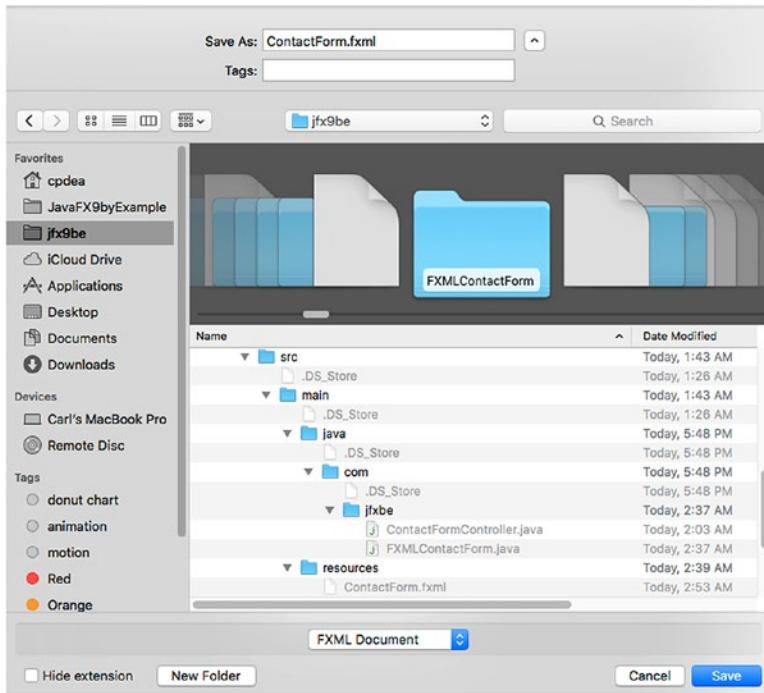


Figure 5-23. The Save As window to save FXML file

Listing 5-8. The ContactForm.fxml File Representing the Contact Form UI View Created Using Scene Builder

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<BorderPane maxHeight="-Infinity" minHeight="-Infinity" minWidth="-Infinity"
prefHeight="400.0" xmlns="http://javafx.com/javafx/8.0.66" xmlns:fx="http://javafx.com/
fxml/1" fx:controller="com.jfxbe.ContactFormController">
    <center>
        <GridPane gridLinesVisible="true" hgap="5.0" vgap="5.0" BorderPane.alignment="CENTER">
            <columnConstraints>
                <ColumnConstraints hgrow="SOMETIMES" maxWidth="-Infinity" minWidth="-Infinity"
prefWidth="100.0" />
                <ColumnConstraints hgrow="ALWAYS" maxWidth="300.0" minWidth="50.0"
prefWidth="150.0" />
            </columnConstraints>
            <rowConstraints>
                <RowConstraints maxHeight="-Infinity" minHeight="-Infinity" prefHeight="30.0"
vgrow="SOMETIMES" />
                <RowConstraints maxHeight="-Infinity" minHeight="10.0" prefHeight="30.0"
vgrow="SOMETIMES" />
            </rowConstraints>
        </GridPane>
    </center>
</BorderPane>
```

```

<RowConstraints maxHeight="-Infinity" minHeight="10.0" prefHeight="30.0"
  vgrow="SOMETIMES" />
</rowConstraints>
<BorderPane.margin>
  <Insets />
</BorderPane.margin>
<padding>
  <Insets bottom="5.0" left="5.0" right="5.0" top="5.0" />
</padding>
<children>
  <Label text="First Name" GridPane.halignment="RIGHT" />
  <Label text="Last Name" GridPane.halignment="RIGHT" GridPane.rowIndex="1" />
  <TextField fx:id="firstNameField" maxWidth="300.0" minWidth="50.0"
    prefWidth="150.0" GridPane.columnIndex="1" />
  <TextField fx:id="lastNameField" maxWidth="300.0" minWidth="50.0"
    prefWidth="150.0" GridPane.columnIndex="1" GridPane.rowIndex="1" />
  <Button fx:id="saveButton" mnemonicParsing="false" onAction="#saveContactAction"
    text="Save" GridPane.columnIndex="1" GridPane.halignment="RIGHT" GridPane.
    rowIndex="2" />
</children>
</GridPane>
</center>
</BorderPane>
```

13. Create a Controller class that provides handler code for UI elements. Create a class called `ContactFormController.java` in the same directory as your main application code `FXMLContactForm.java` (see step 19). Enter the code from Listing 5-9 into a file called `ContactFormController.java`. In the code you'll notice `@FXML` annotations used in the code that are prefixed on properties and methods. These annotations allow the JavaFX platform to bind FXML UI elements to controller properties such as `TextFields` and `Buttons`. Methods in controllers can also be prefixed with a FXML annotation to indicate handler code capable of being bound to an action (event) such as a mouse press.

This binding facility is a mechanism called *dependency injection*. The FXML loader will resolve the references during runtime, essentially wiring up UI elements to the handler code in the Controller class. Later on, you will use Scene Builder to associate the Save button with the controller. There you will notice that the Save button's `OnAction` property will invoke the `saveContactAction()` method shown in listing 5-9.

Listing 5-9. The ContactFormController Class Uses the Annotation to Reference UI Elements Within the FXML File

```

package com.jfxbe;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.TextField;
```

```
/**
 *
 */
public class ContactFormController {
    @FXML private TextField firstNameField;
    @FXML private TextField lastNameField;

    @FXML
    protected void saveContactAction(ActionEvent event) {
        System.out.println("Saving the following information: ");
        System.out.println("First Name: " + firstNameField.getText());
        System.out.println("Last Name: " + lastNameField.getText());
    }
}
```

14. Assign a Controller class to the `ContactForm.fxml` through the Scene Builder tool. Assuming you've loaded the previously created UI, navigate to the Controller class field located under the Controller tab, which is under the Document tab on the left side. Enter the fully qualified class name as `com.jfxbe.ContactFormController` (assuming you have a package namespace). Figure 5-24 shows the field to reference the Controller class. Notice just below the Controller class field there is a table called Assigned `fx:id` that consists of assigned `fx:id` entries. The table is currently blank, but in the next step you will be assigning a `fx:id` name for components, such as the first name and last name text field controls (UI elements).

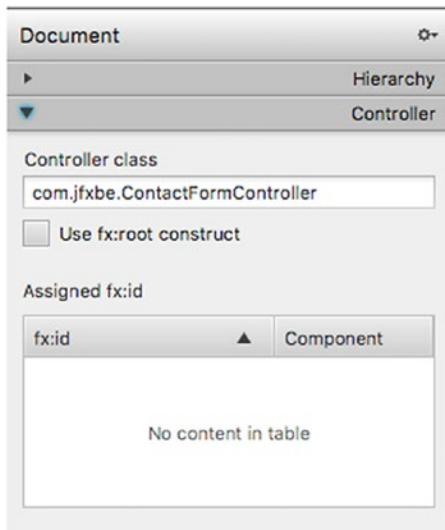


Figure 5-24. Setting the Controller class that references the Contact Form UI (`ContactForm.fxml`)

15. Add `fx:id` name to each UI element in Scene Builder. Assuming you've already referenced the recently created Controller class as `com.jfxbe.ContactFormController`, update the UI element's `fx:id` by selecting the UI control on the canvas or the Hierarchy section. After selecting the control, go to the right side of the Scene Builder tool under the subsection code and enter `firstNameField` into the `fx:id` field property, as shown in Figure 5-25.

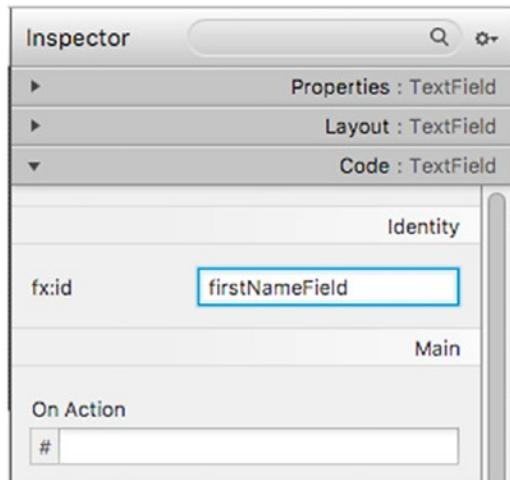


Figure 5-25. Setting the `fx:id` name referencing the first name text field. This will allow the Controller class to access the text field.

16. Repeat Step 15 for the last name field. The last name field's `fx:id` is called `lastNameField`.
17. Bind the action code to the Save button, as seen in Figure 5-26. Similar to Steps 15 and 16, select the Save button's UI element and navigate to its Code subsection to enter an `fx:id` name as `saveButton`. Next, you will set the `OnAction` property to `saveContactAction`. Don't forget to save the UI (FXML). If you recall from Listing 5-9 in Step 13, the `ContactFormController.java` file contains the `saveContactAction()` method.

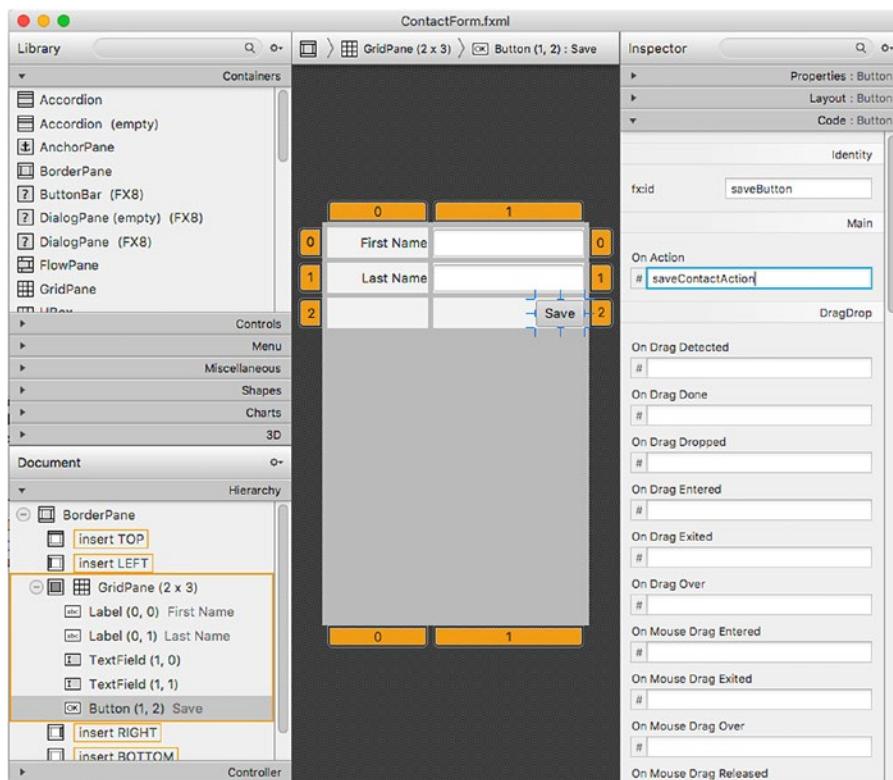


Figure 5-26. Setting the *OnAction* property to invoke the *saveContactAction* function for the Save button

18. Preview the contact form one last time. Figure 5-27 shows a display of the preview window of the ContactForm.fxml file.

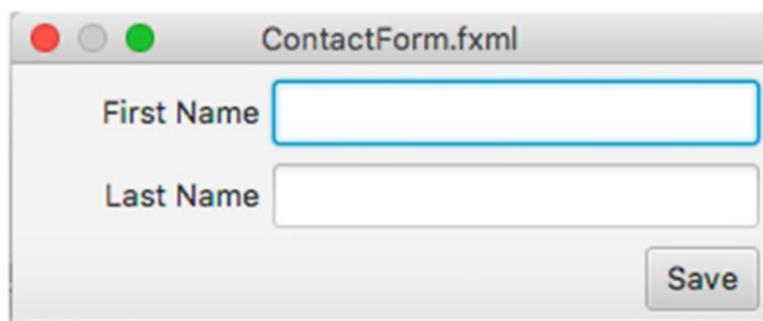


Figure 5-27. Previewing the contact form from the Scene Builder tool

19. Create a main JavaFX application class that will load the FXML to be displayed onto the scene. Listing 5-10 shows the `FXMLContactForm.java` file. If you are coding from scratch, co-locate all three files in the same directory (FXML file, Controller class, and application class).

Listing 5-10. The Main JavaFX FXMLContactForm Application

```
package com.jfxbe;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

import java.io.IOException;

/**
 * The FXMLContactForm application that loads an FXML view
 * to be displayed.
 */
public class FXMLContactForm extends Application {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        stage.setTitle("FXMLContactForm ");

        Parent root = null;
        try {
            root = FXMLLoader.load(getClass().getResource("/ContactForm.fxml"));
        } catch (IOException e) {
            e.printStackTrace();
        }

        Scene scene = new Scene(root, 380, 150, Color.WHITE);
        stage.setScene(scene);
        stage.show();
    }
}
```

20. Build and run the application. Assuming all three files are in the same directory location, navigate or change directory to the files' directory location and then compile and run the code using these commands:

```
$ javac -d . *.java
$ java -cp . com.jfxbe.FXMLContactForm
```

21. The UI should look like similar to Figure 5-28. To test the application, enter a first and last name into the text fields and click on the Save button, then observe the console output.

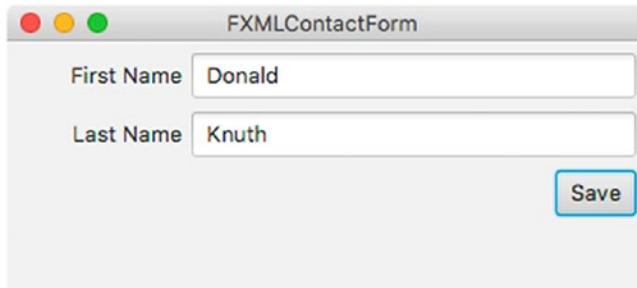


Figure 5-28. The JavaFX FXMLContactForm application

The following is output to the console when the Save button is pressed.

Saving the following information:
 First Name: Donald
 Last Name: Knuth

A Code Walkthrough

By using the Scene Builder tool to create the contact form, you saved the UI into an FXML formatted file. FXML is an XML markup language that represents JavaFX UIs. As the FXML is loaded to be displayed, the XML element tag associated with a JavaFX node is instantiated into Java code dynamically. The main idea of building UIs using FXML is to separate your UI code from your control logic (actions). After graphically laying out your UI elements and saving the file as `ContactForm.fxml`, you created a Controller class file named `ContactFormController.java`. The `ContactFormController` class contains `@FXML` annotations that reference UI elements inside the `ContactForm.fxml` file. The following code snippets are instance variables having the the `@FXML` annotation.

```
@FXML private TextField firstNameField;
@FXML private TextField lastNameField;
```

The `@FXML` annotation is a Java runtime type annotation that tells the `FXMLLoader` utility class which UI elements to inject into the Controller class dynamically. This is JavaFX's dependency injection mechanism that allows controller code to gain access to UI elements. Also, you'll notice `@FXML` annotation can be applied to methods based on UI events such as the `OnAction` property. Listing 5-11 shows the `saveContactAction()` method you saw earlier within the Controller class. Also, recall from Figure 5-27 (Step 17) that the method name is entered into the `OnAction` property field in the Scene Builder tool. This method is invoked when the Save button is pressed.

Listing 5-11. The Handler Code Bound to the Save Button in the UI

```
@FXML
protected void saveContactAction(ActionEvent event) {
    System.out.println("Saving the following information: ");
    System.out.println("First Name: " + firstNameField.getText());
    System.out.println("Last Name: " + lastNameField.getText());
}
```

Lastly, the JavaFX application code loads the FXML file to be displayed. This example treats the FXML file as a resource that will be loaded. Basically, a standard Java application or Java project will typically have a resources directory containing things like property files or images. In this example, we want to put FXML files at the root level. You'll notice the / prefix on the file to denote the file must be located at the root classpath.

The `FXMLLoader.load()` method is given the resource at the root classpath. If you compile your classes into a directory and begin to run the application, you must make sure the FXML file is copied to the root classes directory. See Step 20 to compile and run the application from the command prompt. You set the classpath using `-cp .` and the `.` (dot) denotes the current directory is the root path, thus the `getResource()` method can see the `ContactForm.fxml` file to be loaded.

```
root = FXMLLoader.load(getClass().getResource("/ContactForm.fxml"));
```

Summary

In this chapter you learned about common layouts for creating user interfaces. The first part of the chapter, you briefly examined common layouts such as `HBox`, `VBox`, `FlowPane`, `BorderPane`, and `GridPane`. Using some of the common layout panes such as `HBox` and `Vbox`, you saw detailed code examples and a zoomed-in diagram that described often confusing properties layout nodes have. These properties are spacing, padding, margin, and border widths. The goal was to determine how much space (area) the layout pane along with its children actually occupies. Next, you learned how to create a form-like UI application using the `BorderPane` and `GridPane` nodes. The `GridPane` provided column-, row-, and cell-level constraints. In the form-like UI, you also learned how to control the position and width of child UI elements. An example is that one of the UI elements being controlled was a text field that would squeeze or stretch depending on the user resizing the window.

After learning how to use common layout panes, you learned how to use the powerful Scene Builder tool downloaded from GluonHQ. The chapter also discussed the various features and how to navigate around the tool. After familiarizing with the tool, you got a chance to design a similar form-like UI as the `GridPane` example. The idea of this exercise was to create UI screens graphically instead of programmatically. Once the UI screen was created, you saved the work as a JavaFX FXML file. Learning about separating the UI from control logic, you learned how to use JavaFX's dependency injection mechanism to allow controller code to interact with UI elements. In order to wrap up your understanding of the Scene Builder tool, you learned how to create an application that would load the UI (FXML file) and wire the controller code automatically through the use of the `FXMLLoader` utility class.

CHAPTER 6



User Interface Controls

In the previous chapter, you learned about laying out controls using the *Scene Builder* tool. In this chapter, you learn how to interact with UI controls. Due to space limitations, there aren't code examples for every UI control in the Java 8 and 9 platforms. However, I did create examples with the most common UI controls that you'll typically see in business-type applications.

In Chapter 3's discussion on the JavaFX Scene graph's ability to render basic shape nodes, you learned that shapes extend from the base class `Node` (`javafx.scene.Node`). This chapter explores user interface (UI) controls that also extend from the base class `Node`, but are much more advanced. The main difference is that UI controls are nodes that provide user interactions such as buttons, menus, text fields, sliders, and table views.

Unlike web development using HTML5, JavaFX allows UI control nodes, shape nodes, and canvas nodes to coexist on the Scene graph (draw surface). In contrast, HTML5, text input fields, SVG elements, and canvas elements don't inherit common behaviors like nodes on the JavaFX Scene graph. For example, when using HTML and JavaScript, you cannot easily rotate an HTML button and an SVG rectangle using a common API. When using JavaFX APIs, you can treat UI controls like any other JavaFX node; for example, you can apply animations, transformations, styling, and effects.

JavaFX UI controls are derived classes of the base class `javafx.scene.control.Control` and implement two other interfaces—`javafx.scene.control.Skinnable` and `javafx.event.EventTarget`. To get an in-depth look at how to create *custom controls*, look at Chapter 15 on custom user interfaces. For now, this chapter explores examples on how to use UI controls.

Labels

Labels are read-only controls capable of displaying text and graphics having a background. Labels, unlike the `Text` node, extend from `Control`, as opposed to extending from `Shape`. Labels have a background for text and can append an ellipsis (...) whenever the width of the label (bounding rectangle) is being resized smaller than the text string being displayed to the user. This lets the user know there is more text content to show. Figure 6-1 shows a label that has a very long text string fully displayed (without the ellipses); however, when minimizing the window, the label's width will get reduced, thus showing the truncated text with the ellipses appended on the end, as shown in Figure 6-2.



Figure 6-1. A label with a long text string having the full available width

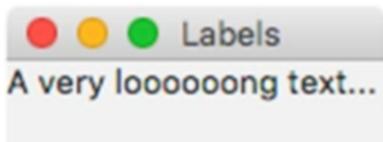


Figure 6-2. A label with a long text string having a reduced width that appends the ellipses denoting there is more text

Labels can also contain a graphic node to be positioned relative to the text. This means that you can add any type of JavaFX node as an image icon. Listing 6-1 shows three possible ways to create JavaFX labels. The first way is to create a label using the empty constructor and subsequently using the `setText()` method. The second way to create a label is using the constructor that accepts a string parameter for the text to be displayed on the label. The last way to create a label is using a constructor that takes two parameters that accept text (String) and a node (Node) for a graphic (icon).

Listing 6-1. Three Ways to Create Labels

```
// Empty text label
Label label1 = new Label();
label1.setText("JavaFX 9 by Example");

// String text label
Label label2 = new Label("JavaFX Rocks!!!!");

// String and Icon label
Node duke = new ImageView(new Image(getClass().getResourceAsStream("/duke.png")));
Label label3 = new Label("JavaFX Rocks!!!", duke);
label3.setContentDisplay(ContentDisplay.TOP);
```

As you can see, the first two methods of creating labels are pretty straightforward; however, the third example gets a little more involved. The third example creates an image node to be used as a graphic (Duke mascot) that is placed above (`ContentDisplay.TOP`) the label's text. Figure 6-3 shows that the label has text and an image that is positioned above the text content of the label.

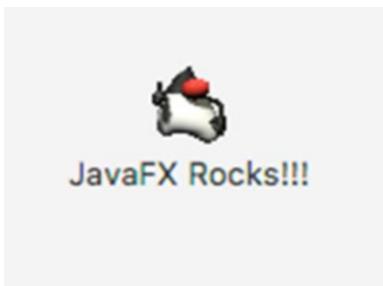


Figure 6-3. A label containing a graphic (image) positioned above (using `ContentDisplay.TOP`) the text. The image of the 3D Duke mascot was created by Joe Palrang (creative commons license).

You will notice from Listing 6-1 the `label3` variable uses the `setContentDisplay()` method that accepts one of the `ContentDisplay` enum values. This method is responsible for how content in the label will be displayed, such as where the graphic is to be positioned, or to show the text content only, etc. Table 6-1 lists all the different ways to display a label's content using the `ContentDisplay` enum constants.

Table 6-1. *ContentDisplay Enum Constants to Specify How Content Can Be Displayed on a Label*

BOTTOM	Content will be placed at the bottom of the label.
CENTER	Content will be placed in the center of the label.
GRAPHIC_ONLY	Only the content will be displayed.
LEFT	Content will be placed to the left of the label.
RIGHT	Content will be placed to the right of the label.
TEXT_ONLY	Only the label's text will be displayed.
TOP	Content will be placed at the top of the label.

Tip ContentDisplay constants used to position content on labels can also be used on JavaFX buttons, check boxes, and menus.

Custom Fonts

Fonts basically style and size text in labels, menus, buttons, and many other controls containing text. Chapter 3 discussed how to apply existing font styles onto Text nodes, but how do you use custom fonts? In JavaFX, you can load standard font file formats such as TrueType fonts (ttf) or OpenType fonts (otf).

Loading a custom font is super easy. First, find the path to the font file you want to load. Second, you will want to initially load the font from the JavaFX application thread. As shown in Listing 6-2, the code loads a TrueType font as a resource from the classpath. You will also notice the overridden JavaFX Application's `init()` method being used to load the custom font. The `init()` method will work off of the JavaFX Application thread before the `start()` method is invoked. The code in Listing 6-2 uses a custom font downloaded from the Google fonts web site at <https://fonts.google.com>. There you can find many *free* type fonts designed and created by many people from all around the world. The following link is where you can download the Kanit font by Cadson Demak.

<https://fonts.google.com/specimen/Kanit?query=kanit&selection.family=Kanit>

Listing 6-2. Overriding the `init()` Method from the JavaFX Application Class to Load a TrueType Font as a Resource from the Classpath

```
@Override
public void init() throws Exception {
    // load custom font
    Font.loadFont(CustomFonts.class.getResource("/Kanit-MediumItalic.ttf")
        .openStream(), 12.0);
}
```

Inside the Application's `start()` method (on the JavaFX application thread), the code can use the previously loaded custom font from the `init()` method. Listing 6-3 shows a label's text that uses the custom font called Kanit with a *MEDIUM* weight having a 60-point font.

Listing 6-3. Assuming the Font Was Previously Loaded, the Code Sets the Custom Font Kanit to Style Text on a Label

```
// Use the custom font
Label labelText = new Label("JavaFX 9 by Example ");
labelText.setFont(Font.font("Kanit", FontWeight.MEDIUM, 60));
```

The output of Listing 6-3 is shown in Figure 6-4.



Figure 6-4. The output of a label using the custom font Kanit

Fonts as Icons

In the next section, you learn how to load custom fonts that are designed to be used as icon images instead of characters. These custom fonts with an image as icons have become common practice for many web sites today. These font packs are scalable vector images used for labels, buttons, and menus or anything needing an image icon. A single Unicode character is basically mapped to an icon image instead of a letter from a Latin alphabet.

Example: Working with Third-Party Font Packs as Icons

In web development you probably have noticed many professional icons used as a graphic applied to labels, buttons, etc., but did you also notice how nicely they are able to be scaled (no pixelation)? This new technique actually takes custom fonts to be represented as vector images. This ingenious idea allows code to retrieve a vector image as if it were an ordinary lettered font to be colorized and resized. If you aren't familiar with the concept, visit the following links:

- FontAwesome: <http://fontawesome.io>
- Weather Icons: <https://erikflowers.github.io/weather-icons>
- Material Design Icons: <http://zavoloklom.github.io/material-design-iconic-font>
- Glyph Icons: <http://getbootstrap.com/components/#glyphicons>

Next, you will see a code example of an application that allows you to search or view icons from these various icon packs. When creating this example, I came across a JavaFX based library called `FontAwesomeFX`, written by Jens Deter, where he takes many of these popular icons to be used in JavaFX applications. By the time you read this chapter, Jens Deter will have likely created things to be modularized for Java 9 (Java 9 modules).

For example, the current implementation of Jens' library has the various icon packs together in one JAR. Here the use of Java 9 modules would allow the library to be broken into different modules, thus allowing applications to only include modules it needs making application packaging much smaller. To get the latest news and updates, visit the [FontAwesomeFX project](https://bitbucket.org/Jerady/fontawesomefx), found at the following link:

<https://bitbucket.org/Jerady/fontawesomefx>

An Example: LabelAwesome a Font Pack Icon Browser App

The LabelAwesome example application allows users to search and browse various font packs from FontAwesome, WeatherIcons, Material Design Icons, and Glyph Icons. The application allows users to type text contained in the name of the icon to be searched. The user can also choose four ways to position the font icon when displaying the labels. The positions to choose from are Top, Bottom, Left, and Right. In Figure 6-5, the LabelAwesome application allows users to search and browse icons from various font packs.

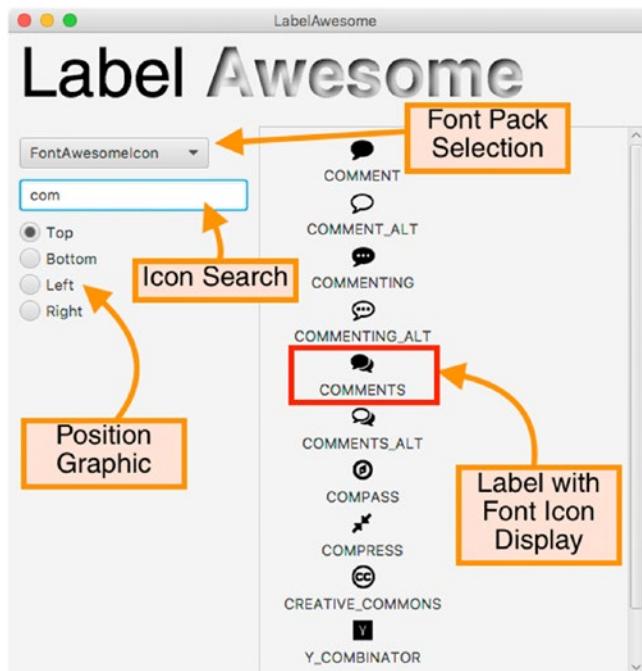


Figure 6-5. The LabelAwesome application allows users to search and browse icons from various font packs. The application also allows the users to choose where to position the font icon in relation to the text on labels.

Getting Started

The LabelAwesome application code resides in the Java file called `LabelAwesome.java`, which is shown in Listing 6-6. But, before we talk about Listing 6-6, let's make sure we include the correct library dependencies. The `fontawesomefx.jar` library must be included on the classpath in order to be successfully *compiled* and *executed*.

If you downloaded the book's source code from Apress publishing, then the `FontAwesomeFX` library is already included. If you have access to Maven or Gradle, you can download it from Maven Central or the Bintray. Maven Central and the Bintray are repositories that host build artifacts such as JAR libraries. For those familiar with Maven, use the dependency's coordinates in Listing 6-4.

Listing 6-4. The Maven Coordinates for Building the Example LabelAwesome.java

```
<groupId>de.jensd</groupId>
<artifactId>fontawesomefx</artifactId>
<version>8.9</version>
```

For those who are familiar with Gradle, use Listing 6-5 as a compile-time dependency in project's `build.gradle` file.

Listing 6-5. The Gradle Dependency for Building the Example LabelAwesome.java

```
compile group: 'de.jensd', name: 'fontawesomefx', version: '8.9'
```

The following are locations to get the latest versions of `FontAwesomeFX`. As of this writing, version 8.9 of the `FontAwesomeFX` library was used.

- Maven: <http://search.maven.org> (Search for `fontawesomefx`)
- Bintray: <https://bintray.com/jerady/maven/FontAwesomeFX>

Now you should be ready to enter Listing 6-6 into a file called `LabelAwesome.java` to be compiled and executed. The `FontAwesomeFX` application can help you find the right icon within the various packs such as `FontAwesome`, `Weather Icons`, `Material Design Icons`, and `Glyph Icons`.

Listing 6-6. Application That Allows User to Search Available Font Icon Images from the `FontAwesomeFX` Library

```
package com.jfxbe;

import de.jensd.fx.glyphs.GlyphIcons;
import de.jensd.fx.glyphs.GlyphsDude;
import de.jensd.fx.glyphs.fontawesome.FontAwesomeIcon;
import de.jensd.fx.glyphs.fontawesome.FontAwesomeIconView;
import de.jensd.fx.glyphs.materialdesignicons.MaterialDesignIcon;
import de.jensd.fx.glyphs.materialdesignicons.MaterialDesignIconView;
import de.jensd.fx.glyphs.materialicons.MaterialIcon;
import de.jensd.fx.glyphs.materialicons.MaterialIconView;
import de.jensd.fx.glyphs.octicons.OctIcon;
import de.jensd.fx.glyphs.octicons.OctIconView;
import de.jensd.fx.glyphs.weathericons.WeatherIcon;
import de.jensd.fx.glyphs.weathericons.WeatherIconView;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.effect.InnerShadow;
```

```
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.scene.text.TextFlow;
import javafx.stage.Stage;

import java.util.*;

/**
 * LabelAwesome is an example using Jens Deters' FontAwesomeFX library.
 * @author carldea
 */
public class LabelAwesome extends Application {
    // lookup icon packs
    private static Map<String, List<GlyphIcons>> ICON_PACKS_MAP = new HashMap<>();
    @Override
    public void init() throws Exception {
        // load all icons
        Font.loadFont(GlyphsDude.class
            .getResource(FontAwesomeIconView.TTF_PATH).openStream(), 10.0);
        Font.loadFont(GlyphsDude.class
            .getResource(MaterialDesignIconView.TTF_PATH).openStream(), 10.0);
        Font.loadFont(GlyphsDude.class
            .getResource(MaterialIconView.TTF_PATH).openStream(), 10.0);
        Font.loadFont(GlyphsDude.class
            .getResource(OctIconView.TTF_PATH).openStream(), 10.0);
        Font.loadFont(GlyphsDude.class
            .getResource(WeatherIconView.TTF_PATH).openStream(), 10.0);
        // Prepare all icons
        ICON_PACKS_MAP.put("FontAwesomeIcon", Arrays.asList(FontAwesomeIcon.values()));
        ICON_PACKS_MAP.put("MaterialDesignIcon",
            Arrays.asList(MaterialDesignIcon.values()));
        ICON_PACKS_MAP.put("MaterialIcon", Arrays.asList(MaterialIcon.values()));
        ICON_PACKS_MAP.put("OctIcon", Arrays.asList(OctIcon.values()));
        ICON_PACKS_MAP.put("WeatherIcon", Arrays.asList(WeatherIcon.values()));
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }
    @Override
    public void start(Stage stage) {
        stage.setTitle("LabelAwesome ");
        BorderPane root = new BorderPane();
        Scene scene = new Scene(root, 600, 450);
```

```

// Create Title
Text labelText = new Text("Label ");
labelText.setFont(Font.font("Helvetica", FontWeight.EXTRA_LIGHT, 60));
Text awesomeText = new Text("Awesome");
InnerShadow innerShadow = new InnerShadow();
innerShadow.setOffsetX(3.0f);
innerShadow.setOffsetY(3.0f);
awesomeText.setEffect(innerShadow);
awesomeText.setFill(Color.WHITE);
awesomeText.setFont(Font.font("Helvetica", FontWeight.BOLD, 60));
TextFlow title = new TextFlow(labelText, awesomeText);
HBox banner = new HBox(title);
banner.setPadding(new Insets(10, 0, 10, 10));
banner.setPrefHeight(70);
root.setTop(banner);

// Display Icon area
VBox labelDisplayPanel = new VBox(5);
labelDisplayPanel.setAlignment(Pos.CENTER);
ScrollPane scrollPane = new ScrollPane(labelDisplayPanel);
root.setCenter(scrollPane);
scrollPane.setPadding(new Insets(10, 10, 10, 10));

// Select Icons packs (ComboBox)
VBox controlsPanel = new VBox(10);
controlsPanel.setPadding(new Insets(10, 10, 10, 10));
List<String> iconPackList = new ArrayList<>();
iconPackList.add("FontAwesomeIcon");
iconPackList.add("MaterialDesignIcon");
iconPackList.add("MaterialIcon");
iconPackList.add("OctIcon");
iconPackList.add("WeatherIcon");

ObservableList<String> obsIconPackList = FXCollections.observableList(iconPackList);
ComboBox<String> iconPacks = new ComboBox<>(obsIconPackList);
iconPacks.setValue(iconPackList.get(0));
controlsPanel.getChildren().add(iconPacks);

// Input Field (TextField)
TextField inputField = new TextField();
inputField.setPrefWidth(200);
inputField.setPromptText("Search Icon Name");
controlsPanel.getChildren().add(inputField);

// Selecting the Icon Position (RadioButton)
VBox imagePositionPanel = new VBox(5);
ToggleGroup position = new ToggleGroup();
RadioButton topPosition = new RadioButton("Top");
topPosition.setSelected(true);
topPosition.setUserData(ContentDisplay.TOP);
topPosition.requestFocus();
topPosition.setToggleGroup(position);

```

```
RadioButton bottomPosition = new RadioButton("Bottom");
bottomPosition.setUserData(ContentDisplay.BOTTOM);
bottomPosition.setToggleGroup(position);

RadioButton leftPosition = new RadioButton("Left");
leftPosition.setUserData(ContentDisplay.LEFT);
leftPosition.setToggleGroup(position);

RadioButton rightPosition = new RadioButton("Right");
rightPosition.setUserData(ContentDisplay.RIGHT);
rightPosition.setToggleGroup(position);

imagePositionPanel.getChildren()
    .addAll(topPosition,
            bottomPosition,
            leftPosition,
            rightPosition);
controlsPanel.getChildren()
    .add(imagePositionPanel);

root.setLeft(controlsPanel);

// As the user types the text is searched.
inputField.textProperty().addListener((o, oldVal, newVal) ->
    showIconList(newVal,labelDisplayPanel,
                iconPacks.getValue(),
                position.getSelectedToggle()
                    .getUserData())
);

// When the radio button select Position to place the Icon
position.selectedToggleProperty().addListener((o, oldVal, newVal) ->
    showIconList(inputField.getText(),
                labelDisplayPanel,
                iconPacks.getValue(),
                position.getSelectedToggle()
                    .getUserData()));

// When Combo box chooses an Icon pack.
iconPacks.setOnAction(actionEvent ->
    showIconList(inputField.getText(),
                labelDisplayPanel,
                iconPacks.getValue(),
                position.getSelectedToggle()
                    .getUserData()));

stage.setScene(scene);
stage.show();
```

```

// Initial display of the current list of Icons
showIconList(inputField.getText(),
    labelDisplayPanel,
    iconPacks.getValue(),
    position.getSelectedToggle()
        .getUserData());
}

private void showIconList(String textInput,
    VBox labelDisplayPanel,
    String iconPack,
    Object position) {

    // Clear the right display
    labelDisplayPanel.getChildren().clear();
    // Obtain the icon pack's list of names.
    List<GlyphIcons> iconPackIcons = ICON_PACKS_MAP.get(iconPack);

    iconPackIcons.stream()
        .filter(iconEnum -> iconEnum.toString().toUpperCase()
            .indexOf(textInput.toUpperCase()) > -1)
        .forEach(iconEnum -> {
            // create a text node using the vector font.
            Text iconShape = new Text(iconEnum.characterToString());
            iconShape.getStyleClass().add("glyph-icon");
            iconShape.setStyle(
                String.format("-fx-font-family: %s; -fx-font-size: %s;",
                    iconEnum.getFontFamily(), 20));
            Label label = new Label(iconEnum.toString(), iconShape);
            label.setContentDisplay((ContentDisplay)position);
            labelDisplayPanel.getChildren().add(label);
        });
}
}
}

```

Compiling the Example

After entering Listing 6-6, you will need to compile it to be later executed. The following shows the commands needed to compile the `LabelAwesome.java` code.

```
$ mkdir classpath
$ javac -cp lib/fontawesomefx-8.9.jar -d classpath src/com/jfxbe/LabelAwesome.java
```

Assuming the current project directory contains a `lib` directory consisting of the FontAwesomeFX JAR library, when compiling Java code, the `-cp` denotes the classpath while the `-d` denotes the directory the compiled code will be written to. The directory in this case is called `classpath`. In Java 9 modules, refer to Chapter 2 to set the module path, which assumes your source code follows the new conventions for Jigsaw.

Run Example Application

After compiling Listing 6-6, you will want to run the example application from the command line. To run the example, you need to include the FontAwesomeFX library on the classpath, including the compiled code in the `classpath` directory. You'll notice that on UNIX/MacOS/Linux systems, the `:` was used as a separator to append multiple paths for the classpath. On a Windows system, you need to use the `;` as a separator. The following command will run the example application.

```
java -cp classpath:lib/fontawesomefx-8.9.jar com.jfxbe.LabelAwesome
```

After running the example, Figure 6-6 shows the output of the `LabelAwesome` application. The `FontAwesomeFX` library supports the following font icon packs: `FontAwesome`, `Material Design`, `Material Icon set`, `Oct Icon set`, and `Weather Icon set`. To see the latest updates, visit Jens Deter's site at <http://www.jensd.de/wordpress/?tag=fontawesome>.

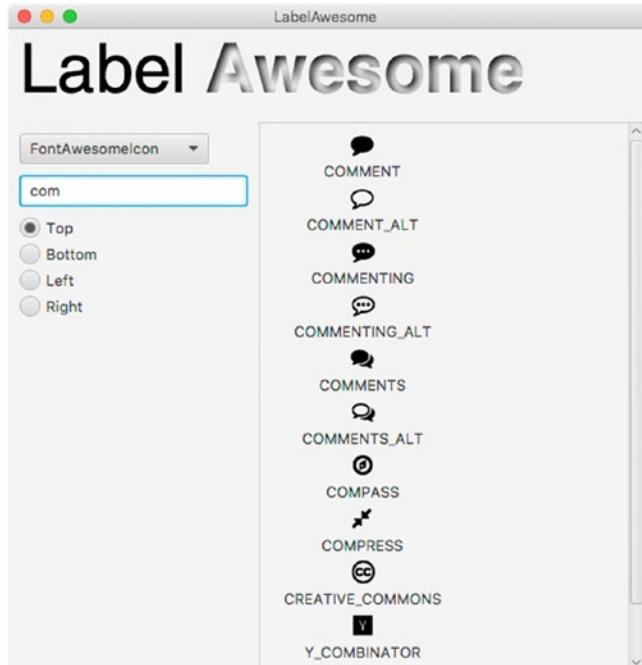


Figure 6-6. The output of Listing 6-2 depicts a sampler application to showcase Jens Deter's `FontAwesomeFX` library

How It Works

In Listing 6-6, the code begins by declaring a static `Map` to hold entries of a String name of the icon pack and a list of `de.jensd.fx.glyphs.GlyphIcons` objects. The following code snippet is the icon packs map declaration.

```
Map<String, List<GlyphIcons>> ICON_PACKS_MAP = new HashMap<>();
```

`GlyphIcons` is an interface for implementation classes that map to individual Unicode characters to a vector icon. For instance, the `FontAwesomeIcon` class is an implementation of the `GlyphIcons` interface, which contains all of the FontAwesome's icon mappings from the TrueType font file `fontawesome-webfont.ttf`. To see the full source code, visit Jens Deter's FontAwesomeFX projects source code at:

<https://bitbucket.org/Jerady/fontawesomex>

The init() Method

Next, the overridden `init()` method (from the `Application` class) implements the loading process of all the supported icon font files based on the path of the font file. The JavaFX `javafx.scene.text.Font` class can load font files from an input stream as a resource on the classpath using the `openStream()` method. In this example, the fonts are loaded using the `loadFont()` method with a given input stream. The font file that is loaded is a TrueType font (TTF) type file that is inside of the `FontAwesomeFX`'s JAR file. The following code loads a font file from the variable (`FontAwesomeIconView.TTF_PATH`), which is a string containing the path of the font at `/de/jensd/fx/glyphs/fontawesome/fontawesome-webfont.ttf`:

```
Font.loadFont(GlyphsDude.class.getResource(FontAwesomeIconView.TTF_PATH).openStream(), 10.0);
ICON_PACKS_MAP.put("FontAwesomeIcon", Arrays.asList(FontAwesomeIcon.values()));
```

The start() Method

After the initial setup of the fonts, the `start()` method sets up the application's interface. It begins by creating the title of the application titled `LabelAwesome`.

Using a TextFlow Control

In Figure 6-5, you'll notice the title's text. `Label` and `Awesome` have different styles but are displayed together. Here the code uses a `javafx.scene.text.TextFlow` node to contain mixed `Text` objects. In this example, the beginning word `Label` is just solid black and the word `Awesome` has an inner shadow effect. The following code snippet adds `Label` and `Awesome` `Text` nodes into the `TextFlow` object.

```
TextFlow title = new TextFlow(labelText, awesomeText);
```

After setting up the title banner, the main display area (border pane's center) is created to contain the display icon area. Here, each label is created with a text description and an icon font (image) to be added to a `VBox` pane to form a row. If you are not familiar to layouts such as `BorderPane` or `VBox`, read Chapter 5, which covers layouts and Scene Builder.

Using a ComboBox Control

Next, you create a combo box selection control containing a string representing the name of a particular font pack. Listing 6-7 is a reshaping of the code section in the `start()` method that creates a combo box control to allow the users to select the supported font packs from the `FontAwesomeFX` library. In Listing 6-7, you'll notice the code setting the `onAction` property using the `setOnAction()` method. This method is where the attached handler (`EventHandler`) gets invoked whenever an icon pack is chosen from the user in a drop-down list. Once the `onAction` event is triggered, the `showIconList()` method is invoked. The `showIconList()` method will be explained in more detail later.

Listing 6-7. Setting Up a Combo Box Control to Allow the User to Select the Font Pack

```
List<String> iconPackList = new ArrayList<>();
iconPackList.add("FontAwesomeIcon");
iconPackList.add("MaterialDesignIcon");
iconPackList.add("MaterialIcon");
iconPackList.add("OctIcon");
iconPackList.add("WeatherIcon");

ObservableList<String> obsIconPackList = FXCollections.observableList(iconPackList);
ComboBox<String> iconPacks = new ComboBox<>(obsIconPackList);
iconPacks.setValue(iconPackList.get(0));

.
// code ommitted (Other controls declared and created)
.

.

// Attaching an action to the combo box control to redisplay the labels.
// When Combo box chooses an Icon pack.
iconPacks.setOnAction(ActionEvent ->
    showIconList(inputField.getText(),
        labelDisplayPanel,
        iconPacks.getValue(),
        position.getSelectedToggle()
            .getUserData())));

```

After creating the combo box control for selecting a font pack, the code creates an auto search text field that allows the users to search through icons within font packs.

Using a TextField Control

Listing 6-8 creates a `TextField` node with a preferred width of 200 pixels and a prompt text property as `Search Icon Name`. When the text field is blank, the prompt text appears in the input text field.

Listing 6-8. Creating an Auto Search Text Field for Searching Icon Names Within a Font Pack

```
// Input Field (TextField)
TextField inputField = new TextField();
inputField.setPrefWidth(200);
inputField.setPromptText("Search Icon Name");

.

.

.

// As the user types the text is searched.
inputField.textProperty().addListener((o, oldVal, newVal) ->
    showIconList(newVal,
        labelDisplayPanel,
        iconPacks.getValue(),
        position.getSelectedToggle()
            .getUserData())));

```

In Listing 6-8, the `textProperty()` of the `inputField` will contain a `ChangeListener` lambda to invoke the private method called `showIconList()` whenever the text changes. The text property changes when the user types text into the text field. The method parameters for the `showIconList()` methods are `textInput`, `labelDisplayPanel`, `iconPack`, and `position`. The following is the `showIconList()` method's signature.

```
private void showIconList(String textInput,
                         VBox labelDisplayPanel,
                         String iconPack,
                         Object position)
```

The `showIconList()` method's first parameter is `textInput`, which is what is entered into the `TextField` control. The `labelDisplayPanel` parameter is a `VBox` container to hold rows of `Label` objects. The next parameter is the `iconPack` parameter of type `String` of the icon pack's name, which is also the key into the `ICON_PACKS_MAP` map object. Lastly, the parameter `position` of type `Object` will be casted into a `ContentDisplay` enum constant to position the icon in relation to the label text.

Continuing with the `start()` method, the code creates radio buttons that allow the users to select where to position the icon relative to the text inside the label. Reshown in Listing 6-9 is the creation of the radio buttons for positioning icons. A `ChangeListener` is attached to the `ToggleGroup` variable `position`. Again, the `ChangeListener` lambda code will invoke the `showIconList()` method given the position.

Listing 6-9. Radio Buttons to Choose the Positioning of the Font Icon When Displaying Labels

```
ToggleGroup position = new ToggleGroup();
.
.

// When the radio button select Position to place the Icon
position.selectedToggleProperty().addListener((o, oldVal, newVal) ->
    showIconList(inputField.getText(),
                 labelDisplayPanel,
                 iconPacks.getValue(),
                 position.getSelectedToggle()
                     .getUserData())
);
```

You'll notice a common theme with all the event handlers and listeners that ultimately make calls to the private method `showIconList()`. This method simply updates the row of labels on the right side of the display.

The Private `showIconDisplay()` Method

Listing 6-10 is a reshowing of the method `showIconDisplay()` from the full listing from Listing 6-6. This method is responsible for refreshing the list of labels displayed on the right side of the screen.

Listing 6-10. The `showIconDisplay()` Method Iterates Over the List of Icons to Be Positioned and Displayed

```
// Clear the right display
labelDisplayPanel.getChildren().clear();

// Obtain the icon pack's list of names.
List<GlyphIcons> iconPackIcons = ICON_PACKS_MAP.get(iconPack);
```

```

iconPackIcons.stream()
    .filter(iconEnum -> iconEnum.toString().toUpperCase()
        .indexOf(textInput.toUpperCase()) > -1)
    .forEach(iconEnum -> {
        // create a text node using the vector font.
        Text iconShape = new Text(iconEnum.characterToString());
        iconShape.getStyleClass().add("glyph-icon");
        iconShape.setStyle(
            String.format("-fx-font-family: %s; -fx-font-size: %s;",
            iconEnum.getFontFamily(), 20));
        Label label = new Label(iconEnum.toString(), iconShape);
        label.setContentDisplay((ContentDisplay) position);
        labelDisplayPanel.getChildren().add(label);
    });
}

```

The code begins by clearing the label display area (VBox) container. The code then obtains the list of GlyphIcons based on the selected icon pack. Next, the code will filter the iconPackIcons list by finding any parts of the search text string contained within any of the icon names. Once filtered, the list of icons will be applied to labels displayed in the VBox container as rows.

Labels and custom fonts are fun, but because of their read-only nature, let's look at buttons.

Buttons

If the JavaFX label is the most basic UI control, the next most basic UI control is the button control. Buttons respond to a mouse press or pressing the spacebar when a button has focus. In JavaFX there are other controls that act like buttons. In fact, all concrete button-like controls extend from the parent javafx.scene.control.ButtonBase class. The following is a list of JavaFX button controls:

- Button
- CheckBox
- Hyperlink
- Radio button
- Toggle button
- Menu/MenuItem

Button

The standard button control can contain text or a graphic (node) similar to the label control explained earlier. Buttons can take on default behaviors such as when a user presses the Enter or Escape key. In scenarios when creating form applications, an OK button can be the default button and can enable users to press the Enter key or click a Cancel button to accept the Escape key. Dialog windows often follow this pattern. The following code snippet creates three buttons—OK, Cancel, and Easy. The OK and Cancel buttons will be configured to accept the Enter and Escape key presses, respectively. The Easy button is set up to invoke the method called doSomethingCool() when pressed.

```
Button okButton = new Button("OK");
okButton.setDefaultButton(true);

Button cancelButton = new Button("Cancel");
cancelButton.setCancelButton(true);

Button easyButton = new Button("Easy");
easyButton.setOnAction( actionEvent -> doSomethingCool() );
```

Check Box

A check box button control is a toggle type button that contains three states—checked, unchecked, and undefined. When a check box indicates a checked state, the control will visibly show a checkmark inside of a box; otherwise, it's unchecked and shows an empty box. The check and unchecked states are controlled by the selected property.

When the state is undefined, the control will display a hyphen mark inside the box. The undefined state is set using the method called `setIndeterminate(true)`. When the indeterminate property is set to true, the selected property is ignored. If you want the check box to allow the users to cycle through the three states, set the `allowIndeterminate` property to true using the `setAllowIndeterminate(true)` method. In Figure 6-7, the three check box states are shown.

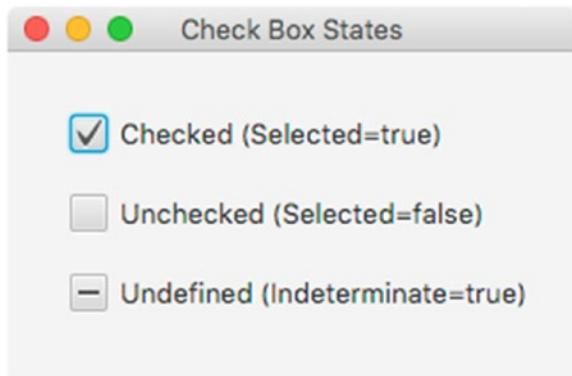


Figure 6-7. The three states a check box control can have

Some use cases involve a tree table view control with a parent node having a check box with an indeterminate box (hyphen mark) with two or more child nodes with at least one of its child nodes being unchecked (selected equaling false) and at least one child node checked.

Note The check box can allow the users to cycle through the three states by setting the `allowIndeterminate` property to true, using the `setAllowIndeterminate(true)` method. In other words, when a user sees an empty check box(`selected=false`), he/she can mouse click to check the check box (`selected=true`), and then click it again to be undefined displayed as a hyphen mark (`indeterminate=true`).

When using check boxes, the most common scenarios usually involve two states: checked and unchecked. For example, when turning the lights on or off. Listing 6-11 shows the code simulating controls to a hypothetical lights control API that turns kitchen lights on or off.

```
CheckBox kitchenLights = new CheckBox("Kitchen Lights");
lightControlManager.kitchenLights()
    .switchProperty()
    .bind(kitchenLights.selectedProperty());
```

Hyperlink

The Hyperlink control is synonymous to the way web pages have text links used for navigating URLs or showing popups. Hyperlinks usually display an underlined text as your mouse pointer hovers over the control. Also, foreground colors can change based on whether the link has already been clicked on. This has the idea of a link that has already been *visited* (the visited property). Listing 6-11 sets up a hyperlink with action code to respond when clicked. The action code will display the link's text or URL as a web page that will be rendered inside a JavaFX WebView node.

Listing 6-11. A Hyperlink with Action Code that Loads a Web Page into a WebView Control

```
Hyperlink searchLink = new Hyperlink("www.google.com");
WebView browser = new WebView();
searchLink.setOnAction( actionEvent ->
    browser.getEngine()
        .load("https://" + searchLink.getText()));
```

Radio Button

Radio buttons are typically used in applications asking the users for one choice among available (two or more) choices. Figure 6-8 shows radio buttons used to allow the users to select a payment type.

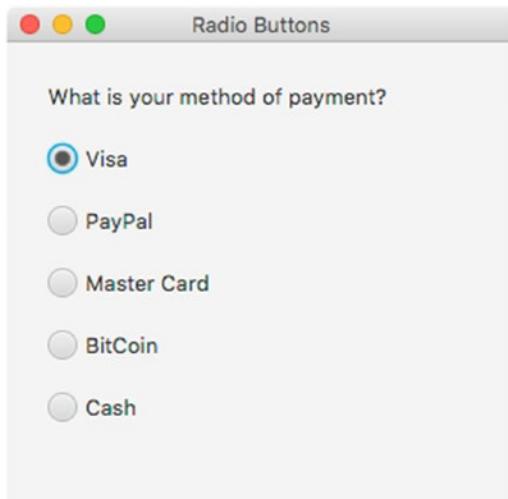


Figure 6-8. A hypothetical payment selection screen. Radio buttons are used to allow the users to select a method of payment.

To create radio buttons as shown in Figure 6-8, look at the code found in Listing 6-12. It begins by creating a ToggleGroup that will hold the chosen value. The toggle group allows only one radio button to be selected at a time. The ToggleGroup has a listener responsible for displaying the payment method to the console when a check box is selected.

Listing 6-12. Code That Asks Users to Select One Payment Method

```
Label questionLabel = new Label("What is your method of payment?");
ToggleGroup group = new ToggleGroup();
RadioButton visaButton = new RadioButton("Visa");
visaButton.setUserData("Visa");
visaButton.setSelected(true);

RadioButton payPalButton = new RadioButton("PayPal");
payPalButton.setUserData("PayPal");

RadioButton masterCardButton = new RadioButton("Master Card");
masterCardButton.setUserData("Master Card");

RadioButton bitCoinButton = new RadioButton("BitCoin");
bitCoinButton.setUserData("BitCoin");

RadioButton cashButton = new RadioButton("Cash");
cashButton.setUserData("Cash");

visaButton.setToggleGroup(group);
payPalButton.setToggleGroup(group);
masterCardButton.setToggleGroup(group);
bitCoinButton.setToggleGroup(group);
cashButton.setToggleGroup(group);

group.selectedToggleProperty().addListener( listener -> {
    System.out.println("Payment type: " + group.getSelectedToggle().getUserData());
});
```

Tip You can use a JavaFX node's set/getUserData() method to put any object into it. In this scenario, the RadioButton controls are set with a String value to uniquely identify each selected item.

Now that you know how to set up labels and buttons, let's have more fun with JavaFX buttons, using an example called Button Fun.

Example: Button Fun

Before you look at the example code, I want to thank the indie game developers at Chasers Gaming for supplying me game sprites to be used in this example. To support visit them at the following:

- <https://www.patreon.com/chasersgaming>
- <https://www.facebook.com/Chasersgamingpage>
- Twitter: @denchaser

Button Fun, shown in Figure 6-9, is an application that demonstrates common JavaFX button controls. The application allows users to view and control three kinds of vehicles.

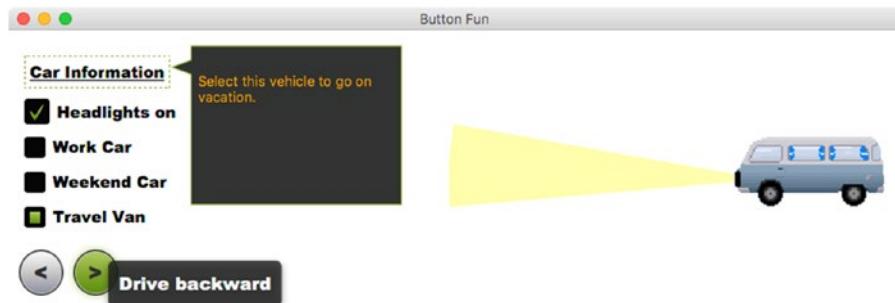


Figure 6-9. Button Fun is an application that allows users to view and control aspects of three vehicle types: a work car, a weekend car, and a travel van.

This example demonstrates the following UI controls:

- Hyperlink
- CheckBox
- RadioButton
- Button
- Popup

Button Fun Instructions

The following are instructions for the Button Fun application:

- *Car Information:* Click on the hyperlink to pop up information about the vehicle you chose.
- *Turn headlights On or Off:* Click on the check box control.
- *Select Vehicle type:* Click to select one of three radio controls of vehicle types—Work Car, Weekend Car, and Travel Van.
- Drive car forward: Click the button with the < arrow.
- Drive car backward: Click the button with the > arrow.

Source Code of ButtonFun.java

Before beginning the example, you will need resources such as image files to be properly loaded during runtime. In order to obtain these resources, visit Apress to download the code example or visit <https://github.com/carldea/jfx9be/tree/master/chap06>. The source code will also contain scripts such as *clean*, *compile*, and *build*. The build script will properly copy the resources directory into the compiled directory (classes).

Listing 6-13 will invoke the *init()* method to load the image sprites to be used in the application. The rest of the code is basically laying out all the controls and wiring up logic or action code to respond to the various button clicks. The CSS styling of the UI is in Listing 6-14, and it will colorize buttons and change properties based on pseudo-classes (states).

Listing 6-13. ButtonFun.java Source Code

```
/**
 * Button Fun is an app showing typical buttons used in application development.
 * Car sprites are from http://opengameart.org/users/chasersgaming
 * @author carldea
 */
public class ButtonFun extends Application {
    private Car[] myCars;
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void init() throws Exception {
        super.init();
        myCars = new Car[3];

        Car workCar = buildCar("/spr_bluecar_0_0.png", "/spr_bluecar_0_0-backwards.png",
                "Select this car to drive to work.");
        Car sportsCar = buildCar("/sportscar.png", "/sportscar-backwards.png",
                "Select this car to drive to the theater.");
        Car travelVan = buildCar("/travel_vehicle.png", "/travel_vehicle.png",
                "Select this vehicle to go on vacation.");

        myCars[0] = workCar;
        myCars[1] = sportsCar;
        myCars[2] = travelVan;
    }

    private Car buildCar(String carForwardFile, String carBackwardFile, String description) {
        Image carGoingForward = new Image(carForwardFile);
        Image carBackward = new Image(carBackwardFile);
        return new Car(carGoingForward, carBackward, description);
    }
}
```

```
@Override
public void start(Stage stage) {

    stage.setTitle("Button Fun");

    BorderPane root = new BorderPane();
    root.setId("background");

    Scene scene = new Scene(root, 900, 250);

    // load JavaFX CSS style
    scene.getStylesheets()
        .add(getClass().getResource("/button-fun.css")
            .toExternalForm());
    VBox leftControlPane = new VBox(10);

    leftControlPane.setPadding(new Insets(0, 10, 20, 15));

    // Create radio buttons for linear,
    // ease in and ease out
    ToggleGroup group = new ToggleGroup();

    RadioButton easeLinearBtn = new RadioButton("Work Car");
    easeLinearBtn.setUserData(myCars[0]);
    easeLinearBtn.getStyleClass().add("option-button");
    easeLinearBtn.setSelected(true);
    easeLinearBtn.setToggleGroup(group);

    RadioButton easeInBtn = new RadioButton("Weekend Car");
    easeInBtn.setUserData(myCars[1]);
    easeInBtn.getStyleClass().add("option-button");
    easeInBtn.setToggleGroup(group);

    RadioButton easeOutBtn = new RadioButton("Travel Van");
    easeOutBtn.setUserData(myCars[2]);
    easeOutBtn.getStyleClass().add("option-button");
    easeOutBtn.setToggleGroup(group);

    // hyperlink
    Hyperlink carInfoLink = createHyperLink(group);

    leftControlPane.getChildren().add(carInfoLink);

    // Create check boxes to turn lights on or off.
    CheckBox headLightsCheckBox = new CheckBox("Headlights on");

    leftControlPane.getChildren().add(headLightsCheckBox);

    leftControlPane.setAlignment(Pos.BOTTOM_LEFT);
    leftControlPane.getChildren().addAll(easeLinearBtn, easeInBtn, easeOutBtn);
```

```

// Create button controls to move car forward or backward.
HBox hbox = new HBox(10);
Button leftBtn = new Button("<");
leftBtn.getStyleClass().add("nav-button");
Button rightBtn = new Button(">");
rightBtn.getStyleClass().add("nav-button");
FlowPane controlPane = new FlowPane();
FlowPane.setMargin(hbox, new Insets(0, 5, 10, 10));
hbox.getChildren().addAll(leftBtn, rightBtn);
controlPane.getChildren().add(hbox);
root.setBottom(controlPane);

// Draw the ground surface
AnchorPane surface = new AnchorPane();
root.setCenter(surface);

root.setLeft(leftControlPane);
int x1 = 20, x2 = 500;
int y1 = 100, y2 = 100;

ImageView carView = new ImageView(myCars[0].carForwards);
carView.setPreserveRatio(true);
carView.setFitWidth(150);
carView.setX(x1);

Arc carHeadlights = new Arc();
carHeadlights.setId("car-headlights-1");
carHeadlights.setCenterX(50.0f);
carHeadlights.setCenterY(90.0f);
carHeadlights.setRadiusX(300.0f);
carHeadlights.setRadiusY(300.0f);
carHeadlights.setStartAngle(170.0f);
carHeadlights.setLength(15f);
carHeadlights.setType(ArcType.ROUND);
carHeadlights.visibleProperty().bind(headLightsCheckBox.selectedProperty());

// Easing car (sports car)
AnchorPane.setBottomAnchor(carView, 20.0);
AnchorPane.setBottomAnchor(carHeadlights, 20.0);
AnchorPane carPane = new AnchorPane(carHeadlights, carView);

AnchorPane.setBottomAnchor(carPane, 20.0);
surface.getChildren().add(carPane);

// The animation based on the currently selected radio buttons.
TranslateTransition animateCar = new TranslateTransition(Duration.millis(400),
carPane);
animateCar.setInterpolator(Interpolator.LINEAR);
animateCar.toXProperty().set(x2);
//animateCar.setInterpolator((Interpolator) group.getSelectedToggle().
getUserData());
animateCar.setDelay(Duration.millis(100));

```

```

// Go forward (Left)
leftBtn.setTooltip(new Tooltip("Drive forward"));
leftBtn.setOnAction( ae -> {
    animateCar.stop();
    Car selectedCar = (Car) group.getSelectedToggle().getUserData();
    carView.setImage(selectedCar.carForwards);
    animateCar.toXProperty().set(x1);
    animateCar.playFromStart();

});

// Go backward (Right)
rightBtn.setTooltip(new Tooltip("Drive backward"));
rightBtn.setOnAction( ae -> {
    animateCar.stop();
    Car selectedCar = (Car) group.getSelectedToggle().getUserData();
    carView.setImage(selectedCar.carBackwards);
    animateCar.toXProperty().set(x2);
    animateCar.playFromStart();
});

group.selectedToggleProperty().addListener((ob, oldVal, newVal) -> {
    Car selectedCar = (Car) newVal.getUserData();
    System.out.println("selected car: " + selectedCar.carDescription);
    carView.setImage(selectedCar.carForwards);
});

stage.setScene(scene);
stage.show();
}

private Hyperlink createHyperLink(ToggleGroup chosenCarToggle) {
    Hyperlink carInfoLink = new Hyperlink("Car Information");
    Popup carInfoPopup = new Popup();
    carInfoPopup.getScene().getStylesheets()
        .add(getClass().getResource("/button-fun.css")
            .toExternalForm());

    carInfoPopup.setAutoHide(true);
    carInfoPopup.setHideOnEscape(true);
    Arc pointer = new Arc(0, 0, 20, 20, -20, 40);
    pointer.setType(ArcType.ROUND);
    Rectangle msgRect = new Rectangle( 18, -20, 200.5, 150);

    Shape msgBubble = Shape.union(pointer, msgRect);
    msgBubble.getStyleClass().add("message-bubble");

    TextFlow textMsg = new TextFlow();
    textMsg.setPrefWidth(msgRect.getWidth() - 5);
    textMsg.setPrefHeight(msgRect.getHeight() - 5);
    textMsg.setLayoutX(pointer.getBoundsInLocal().getWidth() + 5);
    textMsg.setLayoutY(msgRect.getLayoutY() + 5);
}

```

```

Text descr = new Text();
descr.setFill(Color.ORANGE);
textMsg.getChildren().add(descr);

// whenever a selected car set the text.
chosenCarToggle.selectedToggleProperty().addListener((obs, oldVal, newVal) -> {
    Car selectedCar = (Car) newVal.getUserData();
    descr.setText(selectedCar.carDescription);
});

carInfoPopup.getContent().addAll(msgBubble, textMsg);

carInfoLink.setOnAction(actionEvent -> {
    Bounds linkBounds = carInfoLink.localToScreen(carInfoLink.getBoundsInLocal());
    carInfoPopup.show(carInfoLink, linkBounds.getMaxX(), linkBounds.getMinY() -10);
});
return carInfoLink;
}
}

class Car {

String carDescription;
Image carBackwards;
Image carForwards;

public Car(){};

public Car(Image carForwards, Image carBackwards, String desc) {
    this.carForwards = carForwards;
    this.carBackwards = carBackwards;
    this.carDescription = desc;
}
}

```

Listing 6-14. The button-fun.css File Contains JavaFX CSS Styling Attribute to Style the Button Fun Application Screen

```

#background {
    /* button gradient from top to bottom*/
    -light-white: #fbfbfb;
    -light-gray: #c8c8c8;
    -dark-gray: #9090a1;
    -light-green: #aecf64;
    -mid-green: #85ab32;
    -dark-green: #609010;
    -fx-background-color: #3b3b43;
}

```

```
#car-headlights-1 {
    -fx-fill: rgba(255,255,153, 0.8);
}

.nav-button {
    -fx-background-color:
        #000000,
        rgba(255,255,255, 0.8),
        linear-gradient(-light-white, -light-gray, -dark-gray);

    -fx-background-insets: 0,1,2;
    -fx-background-radius: 50%;
    -fx-font-family: 'Arial Black';
    -fx-font-weight: bold;
    -fx-font-size: 20px;
}

.nav-button:hover {
    -fx-background-color:
        #000000,
        rgba(255,255,255, 0.8),
        linear-gradient(-light-green, -mid-green, -dark-green);
    -fx-effect: dropshadow( gaussian , -light-green , 10, 0.0 , 0 , 0 );
}

.nav-button:pressed {
    -fx-background-color:
        #000000,
        rgba(255,255,255, 0.8),
        linear-gradient(-dark-green, -mid-green, -light-green);
}

.option-button .dot {
    -fx-background-radius: 0%;
}

.radio-button .text {
    -fx-text-fill: -fx-text-background-color;
}

.radio-button .radio {
    -fx-background-color: rgba(0,0,0, 0.0), #000000, #000000, #090909;
    -fx-background-insets: 0 0 -1 0, 0, 1, 2;
    -fx-background-radius: 3; /* large value to make sure this remains circular */
    -fx-padding: 0.333333em; /* 4 -- padding from outside edge to the inner black dot */
}

.radio-button:selected .dot {
    -fx-background-color: linear-gradient(-light-green, -mid-green, -dark-green);
    -fx-background-insets: 0 0 -1 0;
}
```

```
.option-button {
    -fx-background-radius: 0%;
    -fx-font-family: 'Arial Black';
    -fx-text-fill: #ffffff;
    -fx-font-weight: bold;
    -fx-font-size: 15px;
}

.check-box {
    -fx-background-radius: 0%;
    -fx-font-family: 'Arial Black';
    -fx-text-fill: #ffffff;
    -fx-font-weight: bold;
    -fx-font-size: 15px;
}

.check-box .box {
    -fx-background-color: rgba(0,0,0, 0.0), #000000, #000000, #090909;
    -fx-background-insets: 0 0 -1 0, 0, 1, 2;
    -fx-background-radius: 3; /* large value to make sure this remains circular */
    -fx-padding: 0.333333em; /* 4 -- padding from outside edge to the inner black dot */
}

.check-box:selected .mark {
    -fx-background-color: linear-gradient(-light-green, -mid-green, -dark-green);
    -fx-background-insets: 0 0 -1 0;
}

.hyperlink {
    -fx-background-radius: 0%;
    -fx-font-family: 'Arial Black';
    -fx-text-fill: #ffffff;
    -fx-font-weight: bold;
    -fx-font-size: 15px;
    -fx-border-color: -mid-green;
}

.message-bubble {
    -fx-fill: rgba(0,0,0, .8);
    -fx-stroke: #85ab32;
}
```

How It Works

- Overriding the `init()` method from the JavaFX Application class to load an array of `Car` objects. Each `Car` instance has an image for the car going forward, an image of the car going backward, and a string description of the vehicle.
- In the `start()` method, the code begins setting up the scene by loading the `button-fun.css` file from Listing 6-14.
- Next, the code creates a toggle group and radio buttons to support the selection of a vehicle.

- After creating radio buttons, the code creates a hyperlink to allow the users to display a popup displaying the selected car's description. The code makes a call to a private method that I created with the following signature:

```
private Hyperlink createHyperLink(ToggleGroup chosenCarToggle)
```

- By passing in the `chosenCarToggle` parameter, a popup can determine which vehicle type was selected to display the correct description.
- Inside of the `createHyperLink()` method, when creating a popup, the code will create a custom shape created by a union of two shapes (an arc and a rectangle). Figures 6-10 depicts both shapes prior to the union. In Figure 6-11, two shapes form the message popup box.

```
Arc pointer = new Arc(0, 0, 20, 20, -20, 40);
pointer.setType(ArcType.ROUND);
Rectangle msgRect = new Rectangle( 18, -20, 200.5, 150);
Shape msgBubble = Shape.union(pointer, msgRect);
```

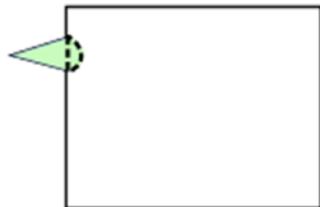


Figure 6-10. An arc and a rectangle shape positioned before the union operation



Figure 6-11. An arc and a rectangle shape positioned after the union operation

- The popup displayed will be positioned just to the right of the hyperlink itself. The following code snippet obtains the bounding box location and converts them to screen coordinates. A popup is displayed in another window (Stage).

```
carInfoLink.setOnAction(actionEvent -> {
    Bounds linkBounds = carInfoLink.localToScreen
        (carInfoLink.getBoundsInLocal());
    carInfoPopup.show(carInfoLink, linkBounds.getMaxX(),
        linkBounds.getMinY() -10);
});
```

- After creating the hyperlink to display the car information, we return to the `start()` method. The code creates a check box control to toggle the headlights of the car. This code creates an arc with a color fill of a translucent yellow.
- Lastly, the `start()` method code creates two buttons that animate the selected car to drive forward or backward. A `TranslateTransition` animation is used to animate the car image. You will learn about animation in Chapter 7, which covers JavaFX graphics.

Menus

Menus are a standard way for windowed desktop applications to allow users to select options. For example, applications typically have a File menu offering options (menu items) to save or open a file. In a windowed environment the users use their mouse to navigate and select menu items. Menus and menu items typically also have the functionality of key combinations to select options, also known as keyboard shortcuts. In other words, key combinations allow quick menu selections without the need of a mouse.

Creating Menus and Menu Items

Before exploring how to invoke code triggered by selecting menu options, let's look at how to create menus. For starters, you must create a menu bar (`javafx.scene.control.MenuBar`) object to hold one-to-many `javafx.scene.control.Menu` objects. Each `Menu` object is similar to a tree hierachal structure, where `Menu` objects can contain `Menu` and `MenuItem` (`javafx.scene.control.MenuItem`) objects. Thus, a menu may contain other menus as submenus or menus within menus. Figure 6-12 depicts a File menu having a Save menu item.

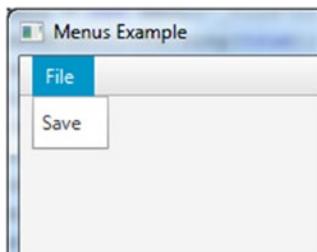


Figure 6-12. A menu bar with a simple File menu that contains a Save menu item

Menu items are child options within a `Menu` object. You can think of `MenuItem` items as leaf nodes (containing no children) in a tree structure. Listing 6-15 is the code to create a menu bar with a File menu that has a Save option as a menu item, as shown in Figure 6-12.

Listing 6-15. Creating a `MenuBar` Having a File Menu with a Save Menu Item

```
MenuBar menuBar = new MenuBar();
Menu fileMenu = new Menu("File");
fileMenu.getItems().add(new MenuItem("Save"));
menuBar.getMenus().add(fileMenu);
```

Although you can create simple menu items like the ones in Figure 6-12, you may want more advanced options such as checked options or radio buttons. Based on the inheritance hierarchy, the following are subclasses of the `MenuItem` class. The following listing shows the available `MenuItem` subclasses to use as menu options. A brief description of each subclass will follow.

- `CheckMenuItem`
- `RadioMenuItem`
- `CustomMenuItem`
- `SeparatorMenuItem`
- `Menu`

A `CheckMenuItem` menu item is similar to a check box UI control, allowing the user to select items optionally. The `RadioMenuItem` menu item is similar to the radio button UI control, allowing the user to select only one item from an item group. In the same manner that you learned about in the prior section on buttons, creating check and radio menus is similar.

Next, the custom menu item, which is a `CustomMenuItem` class, allows developers to create their own specialized menu items. For instance, you may want to have a menu option that behaves like a toggle button.

After that you will see a `SeparatorMenuItem`, which is really a derived class of type `CustomMenuItem`. A `SeparatorMenuItem` is a menu item that displays a visual line to separate menu items.

The last in the bulleted list is the `Menu` class. Since a `Menu` class is a subclass of `MenuItem`, it has a `getItems().add()` method that's capable of adding children, such as other `Menu` and `MenuItem` object instances.

Invoking a Selected MenuItem

Now that you know how to construct menus and menu items, let's see how to invoke code that is attached to each menu item. You'll be happy to know that wiring up handler code to a menu item is done exactly the same way as you wire up JavaFX buttons, which means that menu items also have a `setOnAction()` method. The `setOnAction()` method receives a functional interface of type `EventHandler<ActionEvent>`, which is the handler code that is invoked when the menu item is selected.

Listing 6-16 shows two equivalent implementations of action code to be invoked when the Exit menu item is triggered. The first implementation uses an anonymous inner class, and the second uses Java 8's lambda expressions. A more succinct way to attach an event handler code is to use a lambda functional interface.

Listing 6-16. Implementing Event Handler Code When an Action is Invoked

```
// Implementation that uses an anonymous inner class
exitMenuItem.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent t) {
        Platform.exit();
    }
});

// Implementation that uses lambda / functional interface.
exitMenuItem.setOnAction(ae -> Platform.exit());
```

Example: Working with Menus

To demonstrate various menu selection scenarios, the next code example mimics a security alarm application. As illustrated in Figure 6-13, the menu bar displays three main menus—File, Cameras, and Alarm. When the File menu is selected, it displays three options—New, Save, and Exit—as child menu items.

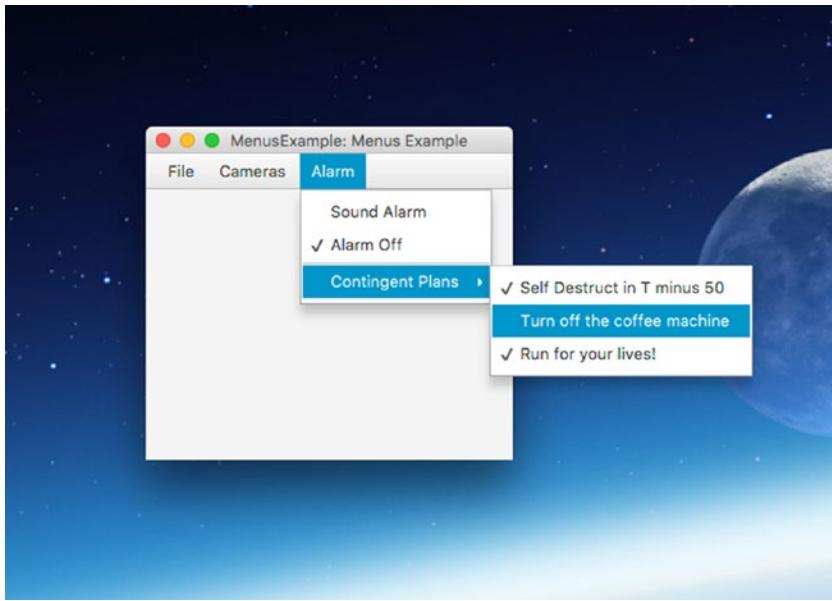


Figure 6-13. Menu example application simulating a security alarm system

Listing 6-17 is from the file `MenuExample.java` and shows the relevant code that demonstrates menu selections. Before executing the code, you may want to know how to use this fictional security alarm application.

Selecting the Cameras menu displays two `CheckMenuItem` menu items, which you can use for the optional selection of Show Camera 1 or Show Camera 2. Lastly, the Alarm menu displays two `RadioMenuItem` options and a submenu Contingent Plans. The submenu displays three child `CheckMenuItem` options.

Listing 6-17. The `MenuExample.java` File Is a Hypothetical Alarm System Application Demonstrating Various Menu Items

```
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("Menus Example");
    BorderPane root = new BorderPane();
    Scene scene = new Scene(root, 300, 250, Color.WHITE);

    MenuBar menuBar = new MenuBar();
    root.setTop(menuBar);
```

```
// File menu - new, save, exit
Menu fileMenu = new Menu("File");

MenuItem newItem = new MenuItem("New");
MenuItem saveMenuItem = new MenuItem("Save");
MenuItem exitMenuItem = new MenuItem("Exit");
exitMenuItem.setOnAction(actionEvent -> Platform.exit() );

fileMenu.getItems().addAll(newItem,
    saveMenuItem,
    new SeparatorMenuItem(),
    exitMenuItem
);

// Cameras menu - camera 1, camera 2
Menu cameraMenu = new Menu("Cameras");

CheckMenuItem cam1MenuItem = new CheckMenuItem("Show Camera 1");
cam1MenuItem.setSelected(true);
cameraMenu.getItems().add(cam1MenuItem);

CheckMenuItem cam2MenuItem = new CheckMenuItem("Show Camera 2");
cam2MenuItem.setSelected(true);
cameraMenu.getItems().add(cam2MenuItem);

// Alarm Menu
Menu alarmMenu = new Menu("Alarm");

// sound or turn alarm off
ToggleGroup tGroup = new ToggleGroup();
RadioMenuItem soundAlarmItem = new RadioMenuItem("Sound Alarm");
soundAlarmItem.setToggleGroup(tGroup);

RadioMenuItem stopAlarmItem = new RadioMenuItem("Alarm Off");
stopAlarmItem.setToggleGroup(tGroup);
stopAlarmItem.setSelected(true);
alarmMenu.getItems().addAll(soundAlarmItem,
    stopAlarmItem,
    new SeparatorMenuItem());

// Contingency Menu options
Menu contingencyPlans = new Menu("Contingent Plans");
contingencyPlans.getItems().addAll(
    new CheckMenuItem("Self Destruct in T minus 50"),
    new CheckMenuItem("Turn off the coffee machine "),
    new CheckMenuItem("Run for your lives! "));

alarmMenu.getItems().add(contingencyPlans);
menuBar.getMenus().addAll(fileMenu, cameraMenu, alarmMenu);
primaryStage.setScene(scene);
primaryStage.show();
}
```

How It Works

The code begins by creating a `MenuBar` control, which may contain one to many menu (`MenuItem`) objects. After the `MenuBar` is created, it is set as the top region on the `BorderPane` layout. Since the top region allows a resizable node to take up the available horizontal space, the `MenuBar` control will stretch to the width of the `BorderPane` (the root scene). Listing 6-18 sets the top region of the `BorderPane` to a `MenuBar`.

Listing 6-18. A Menu Bar Is Set as the Top Region Node of the Root BorderPane

```
MenuBar menuBar = new MenuBar();
BorderPane root = new BorderPane();
root.setTop(menuBar);
```

Next, the code instantiates `Menu` objects that contain one or more menu item (`MenuItem`) objects and other `Menu` objects, making submenus.

After creating the `File` menu, the code creates the `Cameras` menu. This menu will contain `CheckMenuItem` objects. These menu items allow users to optionally select or unselect *Show Camera 1* and *Show Camera 2* as menu items.

Last is the implementation of the `Alarm` menu, which contains two radio menu items (`RadioMenuItem`), a separator (`SeparatorMenuItem`), and a submenu (`Menu`). To create the radio menu items for the `Alarm` menu, the code itself creates an instance of a `ToggleGroup` class. The `ToggleGroup` class is also used on regular radio buttons (`RadioButtons`) to allow one selected option only. Listing 6-19 creates radio menu items (`RadioMenuItem`s) to be added to the `alarmMenu` `Menu` object.

Listing 6-19. Creating Menu Items That Contain Radio and Separator Menu Items

```
// Alarm menu
Menu alarmMenu = new Menu("Alarm");

// Sound or turn alarm off
ToggleGroup tGroup = new ToggleGroup();
RadioMenuItem soundAlarmItem = new RadioMenuItem("Sound Alarm");
soundAlarmItem.setToggleGroup(tGroup);

RadioMenuItem stopAlarmItem = new RadioMenuItem("Alarm Off");
stopAlarmItem.setToggleGroup(tGroup);
stopAlarmItem.setSelected(true);

alarmMenu.getItems().addAll(soundAlarmItem, stopAlarmItem, new SeparatorMenuItem());
```

The code also adds a visual separator, by instantiating a `SeparatorMenuItem` class that is added to a menu via the `getItems().addAll()` method. The method `getItems()` returns an observable list of `MenuItem` objects (`ObservableList<MenuItem>`). You'll learn about `ObservableList` collections later, but briefly, they are collections that have the ability to notify and update UI controls as items are added or removed. For a full description, see the section "The `ObservableList` Collection Class" later in this chapter.

The last child menu item added to the `Alarm` menu is a submenu, called `Contingent Plans`. To add some levity, I've created some humorous menu options for contingency plans when an emergency occurs.

Additional Ways to Select Menus and Menu Items

In the previous example, you learned how to create and invoke menu items using your mouse, but did you know that there are other ways to invoke menu items? In this section, you will learn three additional ways to invoke menu items: by constructing key mnemonics, key combinations, and context menus.

Key Mnemonics

Standard menus generally have keyboard mnemonics to select menu items without using a mouse. Typically, applications that use a menu allow the user to press the Alt key, which puts an underscore () beneath a letter of the menu text label. After the user presses the letter, the menu will drop down to display its child menu items and the user will be allowed to navigate with arrow keys. Keep in mind that this behavior is only on the Windows platform.

To perform this behavior in code, you instantiate a menu by invoking the constructor that receives a String value as before, but this time you place an underscore character preceding the chosen letter in the menu or menu item's text. Also, to let the system recognize the mnemonic, you simply pass true to the setMnemonicParsing(true) method. Listing 6-20 creates a File menu that uses the letter “F” as the mnemonic.

Listing 6-20. Creating a File Menu with a Mnemonic for the Letter “F”

```
MenuItem fileMenu = new MenuItem("_File");
fileMenu.setMnemonicParsing(true);
```

Key Combinations

A key combination is a combination of keystrokes that select a menu option. For instance, most applications on the Windows platform use the key combination Ctrl+S to save a file. On the MacOS platform, the key combination is Command (⌘)+S. Keys such as Ctrl, Command, Alt, Shift, and Meta are called modifier keys. Usually these modifiers are pressed in combination with a single letter. Sometimes key combinations are referred to as keyboard shortcuts (mapped to menu items).

To create a key combination, you need an instance of the KeyCodeCombination object that will contain the keystroke and the modifiers. Listing 6-21 provides a key code combination of (Ctrl or Meta)+S. You will notice that the code uses the KeyCombination.SHORTCUT_DOWN value as the key modifier instead of CONTROL_DOWN or META_DOWN. The reason is simple; the value of SHORTCUT_DOWN will enable the application to be cross-platform. The values CONTROL_DOWN and META_DOWN are system dependent on the Windows and MacOS platforms, respectively, but SHORTCUT_DOWN works on all platforms.

Listing 6-21. A Menu Item Mapped to a Keyboard Shortcut of Ctrl+S (Windows) or Command+S (Mac)

```
MenuItem saveItem = new MenuItem("_Save");
saveItem.setMnemonicParsing(true);
saveItem.setAccelerator(new KeyCodeCombination(KeyCode.S, KeyCombination.SHORTCUT_DOWN));
```

Context Menus

Context menus are popup menus displayed when a user right-clicks the mouse button on a JavaFX UI control or a stage surface. To create a context menu, you need to instantiate a ContextMenu class. Exactly like a regular menu, the ContextMenu menu has a getItems().add() method to add menu items. The following code snippet shows a context menu instantiated with a menu item (exitItem):

```
ContextMenu contextFileMenu = new ContextMenu(exitItem);
```

To respond to a right-click of the mouse on the scene in an application, you can basically add an event handler to listen for a right-click event. Once that is detected, the context menu's `show()` method is invoked. The code in Listing 6-22 sets up an event handler to show and hide a context menu based on a right or left mouse click, respectively. Notice the `hide()` method invoked by the primary mouse click (left-click) to remove the context menu.

Listing 6-22. Attaching an Event Handler That Displays a Popup Menu Upon Encountering a Right-Click Mouse Event on a JavaFX Stage

```
primaryStage.addEventHandler(MouseEvent.MOUSE_CLICKED,
(MouseEvent me) -> {
    if (me.getButton() == MouseButton.SECONDARY || me.isControlDown()) {
        contextFileMenu.show(root, me.getScreenX(), me.getScreenY());
    } else {
        contextFileMenu.hide();
    }
});
```

To see the full code listing showing the use of these additional strategies to invoke menus, visit the book's website to download the project called `KeyCombinationsAndContextMenu.java`.

The ObservableList Collection Class

When using the Java Collections API, you'll notice the many useful container classes that represent data structures such as sets, maps, and lists. One of the commonly used concrete list classes is `java.util.ArrayList`, which implements the `List` interface. For a long time, Java Swing developers built applications using `ListModels` and `ArrayList` to represent a list of objects that will be displayed in a list-like UI control. The issues that caused a lot of grief were the ability to synchronize list models and view components. To solve this problem, you can use JavaFX's `ObservableList` class. The `ObservableList` class is a collection that is capable of notifying UI controls when objects are added, updated, and removed.

Here is how to take an existing collection to be wrapped by an observable list using the `FXCollections.observableArrayList()` function:

```
List<String> doctors = ...// query database
ObservableList<String> obsDoctors = FXCollections.observableArrayList(doctors);
```

JavaFX `ObservableLists` are typically used in list UI controls such as `ListView` and `TableView`. Let's look at two examples that will use the `ObservableList` collection class. To see other observable collections such as maps, sets, and arrays, refer to the Javadoc documentation on the API `javafx.collections.FXCollections`.

Working with ListViews

The JavaFX `ListView` control is similar to the Java Swing `JList` component. A `ListView` is backed with an observable list (`ObservableList`). Similar to Java Swing's `ListModel`, when altering items in an observable list, the `ListView` control will update its UI display automatically. The following code creates an observable list to be used in a `ListView` control.

```
ObservableList<String> javafxers = FXCollections.observableArrayList(
    "Carl Dea",
    "Gerrit Grunwald",
    "Mark Heckler",
    "José Pereda",
    "Sean Phillips");
ListView<String> myListView = new ListView<>(javafxers);
```

This code shows how to create a `ListView` backed by an observable list of `String` objects. By default, the `ListView` will simply display items if they are strings. But what if they contain domain objects that you've created? The answer is by using cell factories. You will learn about them in the next section on `TableViews`; for now let's see how to manipulate items in an observable list and a `ListView` control.

Example: Hero Picker

The Hero Picker example application will demonstrate the `ObservableList`'s capabilities; it displays two JavaFX `ListView` UI controls and allows the user to transfer items between the two `ListView` controls. One list contains candidates for hero status and the other contains those who have already achieved it. As shown in Figure 6-14, this application allows you to choose a candidate to be transferred to the other `ListView` control, which contains the chosen heroes.

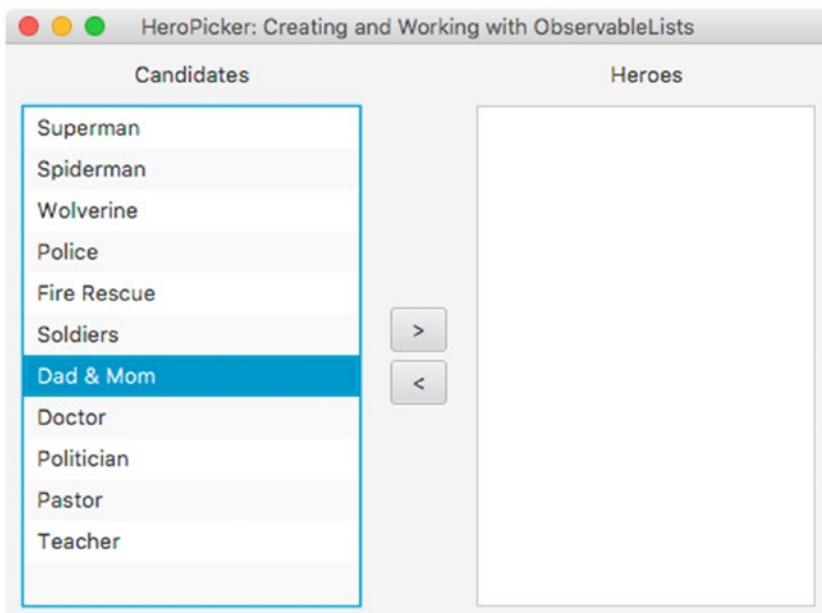


Figure 6-14. Hero Picker application to choose candidates as heroes to demonstrate the `ListView` control

Listing 6-23. The Hero Picker Application Demonstrates the ObservableList and ListView Classes

```
/**
 * Hero Picker application demonstrates observable lists and list views.
 * @author cdea
 */
public class HeroPicker extends Application {

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("HeroPicker: Creating and Working with ObservableLists");
        BorderPane root = new BorderPane();
        Scene scene = new Scene(root, 500, 350, Color.WHITE);

        // create a grid pane
        GridPane gridpane = new GridPane();
        gridpane.setPadding(new Insets(10));
        gridpane.setHgap(10);
        gridpane.setVgap(10);
        gridpane.setPrefHeight(Double.MAX_VALUE);
        ColumnConstraints column1 = new ColumnConstraints(150, 150, Double.MAX_VALUE);
        ColumnConstraints column2 = new ColumnConstraints(50);
        ColumnConstraints column3 = new ColumnConstraints(150, 150, Double.MAX_VALUE);
        column1.setHgrow(Priority.ALWAYS);
        column3.setHgrow(Priority.ALWAYS);
        gridpane.getColumnConstraints().addAll(column1, column2, column3);

        // Candidates label
        Label candidatesLbl = new Label("Candidates");
        GridPane.setAlignment(candidatesLbl, HPos.CENTER);
        gridpane.add(candidatesLbl, 0, 0);

        // Heroes label
        Label heroesLbl = new Label("Heroes");
        gridpane.add(heroesLbl, 2, 0);
        GridPane.setAlignment(heroesLbl, HPos.CENTER);

        // Candidates
        ObservableList<String> candidates = FXCollections.observableArrayList("Superman",
            "Spiderman",
            "Wolverine",
            "Police",
            "Fire Rescue",
            "Soldiers",
            "Dad & Mom",
            "Doctor",
            "Politician",
            "Pastor",
            "Teacher");
        ListView<String> candidatesListView = new ListView<>(candidates);
        gridpane.add(candidatesListView, 0, 1);
    }
}
```

```
// heroes
ObservableList<String> heroes = FXCollections.observableArrayList();
ListView<String> heroListView = new ListView<>(heroes);
gridpane.add(heroListView, 2, 1);

// select heroes
Button sendRightButton = new Button(" > ");
sendRightButton.setOnAction((ActionEvent event) -> {
    String potential = candidatesListView.getSelectionModel().getSelectedItem();
    if (potential != null) {
        candidatesListView.getSelectionModel().clearSelection();
        candidates.remove(potential);
        heroes.add(potential);
    }
});

// deselect heroes
Button sendLeftButton = new Button(" < ");
sendLeftButton.setOnAction((ActionEvent event) -> {
    String notHero = heroListView.getSelectionModel().getSelectedItem();
    if (notHero != null) {
        heroListView.getSelectionModel().clearSelection();
        heroes.remove(notHero);
        candidates.add(notHero);
    }
});

VBox vbox = new VBox(5);
vbox.getChildren().addAll(sendRightButton, sendLeftButton);
vbox.setAlignment(Pos.CENTER);
gridpane.add(vbox, 1, 1);
root.setCenter(gridpane);

GridPane.setVgrow(root, Priority.ALWAYS);
primaryStage.setScene(scene);
primaryStage.show();
}

public static void main(String[] args) {
    launch(args);
}
}
```

Figure 6-15 shows the output from Listing 6-23, the Hero Picker application.

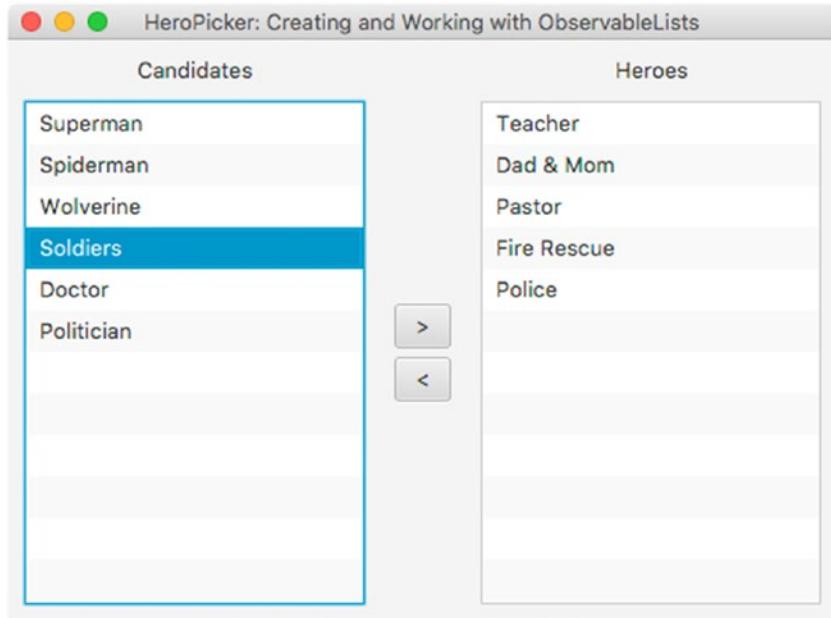


Figure 6-15. The output of the Hero Picker application, demonstrating the use of the `ObservableList` and `ListView` classes

How It Works

Created in this example is a GUI application that allows users to choose their favorite heroes. This is quite similar to system administration applications that manage user roles by adding or removing items from list box components. In JavaFX the code example uses two `ListView` controls to hold `String` objects. For the sake of brevity, we will not discuss the use of the layouts, but jump right into the relevant code relating to the button actions and `ListView` controls.

Based on Listing 6-23, right before creating an instance of a `ListView`, the code creates an `ObservableList` containing the candidates. Here you'll notice the use of a factory class called `FXCollections`, into which you can pass common collection types to be wrapped and returned to the caller as an `ObservableList`. In the example, an array of strings is passed into the `FXCollections.observableArrayList()` method instead of an `ArrayList`. Using the `ObservableList` allows you to update the list of items in the array and in the display on the screen concurrently with a single method call.

Hopefully you get the idea about how to use the `FXCollections` class. The following code line calls the `FXCollections.observableArrayList()` method class to return an observable list (`ObservableList`):

```
ObservableList<String> candidates = FXCollections.observableArrayList(...);
```

After the code creates an `ObservableList`, a `ListView` class is instantiated using a constructor that receives the observable list. The following code creates and populates a `ListView` object:

```
ListView<String> candidatesListView = new ListView<>(candidates);
```

The last item of business is where the code manipulates `ObservableLists` as if they were `java.util.ArrayLists`. As they are manipulated, the `ListView` is notified and automatically updated to reflect the changes of the `ObservableList`. Listing 6-24 implements the action event code when the user presses the Send right (>) button.

Listing 6-24. Action Code to Process Selected Items

```
// select heroes
Button sendRightButton = new Button(">");
sendRightButton.setOnAction( actionEvent -> {
    String potential = candidatesListView.getSelectionModel().getSelectedItem();
    if (potential != null) {
        candidatesListView.getSelectionModel().clearSelection();
        candidates.remove(potential);
        heroes.add(potential);
    }
});
```

To set action code for the button, the code uses a lambda expression with an `actionEvent` (`ActionEvent`) as a parameter, which is invoked when the button is clicked. When a button-press event arrives, the code determines which item in the `ListView` was selected via the `getSelectionModel()` method on the `ListView`. Once the item is determined, the code clears the selection, removes the item, and adds the item to the `Heroes ObservableList`. If you do not clear the selection, the item after the removed item will be selected.

When working with `ObservableList`, having items removed or added will notify the control. What's great about this is that there is no need to refresh the `ListView` UI control when the `ObservableList` changes, because the `ListView` will automatically be updated.

Working with `TableViews`

You can also use `ObservableLists` in another list-like UI controls, like the `TableView`. JavaFX's `javafx.scene.control.TableView` control is analogous to Swing's `JTable` component, which contains rows, columns, and cells very similar to spreadsheet applications. The following code snippet creates a `TableView` control with one column targeting the `firstName` of a domain object.

```
TableView<Person> employeeTableView = new TableView<>();
TableColumn<Person, String> firstNameCol = new TableColumn<>("First Name");
firstNameCol.setCellValueFactory(new PropertyValueFactory<Person, String>("firstName"));
```

What Is a Cell Factory?

Cell factories provide a way to customize how to display content in cells such as row of a `ListView` or a cell in a column of a `TableView` control. Listing 6-25 shows how to create a cell factory to display the first and last name of `Person` object within a `ListView`'s cell row.

Listing 6-25. Setting a Cell Factory (Callback) for a List View Control

```
// display first and last name
personListView.setCellFactory(new Callback<ListView<Person>, ListCell<Person>>() {
    @Override
    public ListCell<Person> call(ListView<Person> param) {
        Label leadLbl = new Label();
        ListCell<Person> cell = new ListCell<Person>() {
            @Override
            public void updateItem(Person item, boolean empty) {
                super.updateItem(item, empty);
                if (item != null) {
                    leadLbl.setText(item.getAliasName());
                    setText(item.getFirstName() + " " + item.getLastName());
                }
            }
        }; // ListCell
        return cell;
    }
}); // setCellFactory
```

Of course, this code looks somewhat verbose, so Listing 6-26 shows converted the code that uses the more concise lambda based syntax.

Listing 6-26. A Cell Factory Set Using the Lambda-Based Syntax to Cut Down on the Boilerplate Code

```
// display first and last name with tooltip using alias
personListView.setCellFactory(listView -> {
    Label leadLbl = new Label();
    ListCell<Person> cell = new ListCell<Person>() {
        @Override
        public void updateItem(Person item, boolean empty) {
            super.updateItem(item, empty);
            if (item != null) {
                leadLbl.setText(item.getAliasName());
                setText(item.getFirstName() + " " + item.getLastName());
            }
        }
    }; // ListCell
    return cell;
}); // setCellFactory
```

Making Table Cells Editable

Did you ever want to be able to modify a cell? JavaFX TableView controls and their cells can be edited. When designing TableViews, the JavaFX engineers had already thought of this and made convenient functions that provide editable cells.

Edit a Table Cell as a TextField

An obvious editable cell would allow the users to edit the information tied to that cell with a `textfield` control. The following code snippet sets up a column to be an editable text field.

```
TableColumn<Person, String> aliasNameCol = new TableColumn<>("Alias");  
aliasNameCol.setCellValueFactory(new PropertyValueFactory<>("aliasName"));  
aliasNameCol.setCellFactory(TextFieldTableCell.forTableColumn());
```

Figure 6-16 shows an editable cell in the Alias column on a TableView control containing employees.

Figure 6-16. An editable cell using a text field to modify employee information. The alias column is set up with a cell factory using the `TextFieldTableCell.forTableColumn()` convenience function.

Edit a Table Cell as a ComboBox

Another control common to editing cells is the combo box. A combo box allows users to select enumerations as a drop-down list selection. The following code snippet sets up a cell column to be a combo box. Figure 6-17 shows the combo box within a cell of the Mood column.

Employees			
Alias	First Name	Last Name	Mood
Wolverine	James	Howlett	Angry
Cyclops	Scott	Summers	Happy
Storm	Ororo	Munroe	Positive ▾
			Happy
			Sad
			Angry
			Positive

Figure 6-17. An editable cell using a combo box to modify an employee's mood property. The mood column is set up with a cell factory with the convenience function `ComboBoxTableCell.forTableColumn()` method.

```
import static com.jfxbe.Person.MOOD_TYPES;

ObservableList<MOOD_TYPES> moods = FXCollections.observableArrayList(MOOD_TYPES.values());

TableColumn<Person, MOOD_TYPES> moodCol = new TableColumn<>("Mood");
moodCol.setCellValueFactory(new PropertyValueFactory<>("mood"));
moodColumn.setCellFactory(ComboBoxTableCell.forTableColumn(moods));
```

Example: Bosses and Employees Working with Tables

The previous example used a list of String objects for the items inside a `ListView`, and in this section we use a *domain object* often called *POJOs* (plain old Java objects). Keep in mind when using your own domain objects in `ListView` and `TableView` controls that you will want to customize the display of object's fields for each row or cell.

As an example of the `TableView` control, I've created an application called *Bosses and Employees* shown in Figure 6-18. The example application demonstrates how to populate a `TableView` containing an observable list of employees (`Person` class) that is based on a selection of a boss from a `ListView` control (left side). In Figure 6-18, you'll also notice the tooltip when hovering over the list view cell (row).

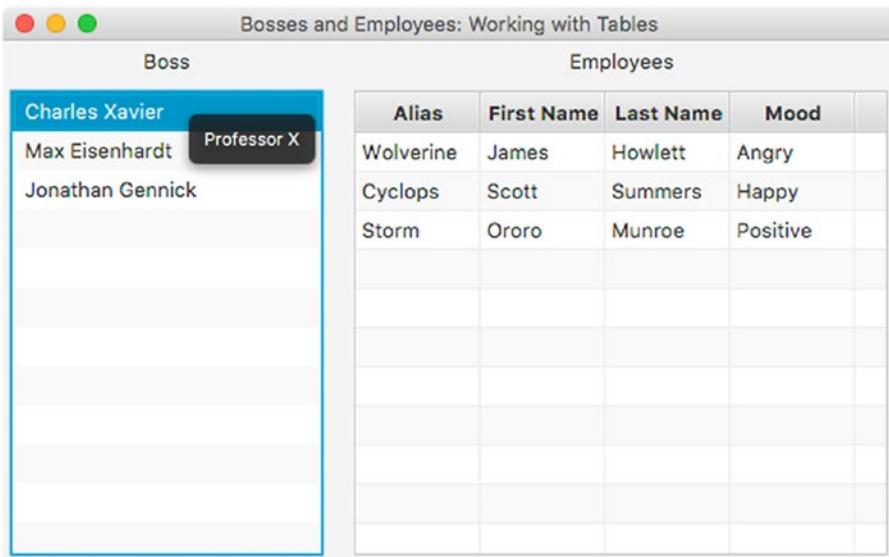


Figure 6-18. The output of the Bosses and Employees application, which demonstrates a JavaFX ListView and TableView control. Observable lists will contain domain objects of the Person class.

Domain Objects

Before delving into the main GUI code, we examine the use of a domain object that represents a person (`Person`) in this case a boss or an employee. Listing 6-27 shows an implementation of a JavaFX bean (`Person` class) that represents a boss or an employee that will be displayed in a `ListView` and `TableView` control. A `Person` instance will contain the following properties: `aliasName`, `firstName`, `lastName`, and `mood`. A person also has an `employees` list to hold subordinate employees (`minions`). The GUI code to handle a list view selection and the capability to populate the table view appears in Listing 6-28.

Listing 6-27. The Person Class as a JavaFX Bean Used in ObservableLists

```
public class Person {  
    public enum MOOD_TYPES {  
        Happy,  
        Sad,  
        Angry,  
        Positive  
    }  
    private StringProperty aliasName;  
    private StringProperty firstName;  
    private StringProperty lastName;  
    private ObjectProperty<MOOD_TYPES> mood;  
  
    private ObservableList<Person> employees = FXCollections.observableArrayList();
```

```
public Person(String alias, String firstName, String lastName, MOOD_TYPES mood) {
    setAliasName(alias);
    setFirstName(firstName);
    setLastName(lastName);
    setMood(mood);
}

public final void setAliasName(String value) {
    aliasNameProperty().set(value);
}

public final String getAliasName() {
    return aliasNameProperty().get();
}

public StringProperty aliasNameProperty() {
    if (aliasName == null) {
        aliasName = new SimpleStringProperty();
    }
    return aliasName;
}

public final void setFirstName(String value) {
    firstNameProperty().set(value);
}

public final String getFirstName() {
    return firstNameProperty().get();
}

public StringProperty firstNameProperty() {
    if (firstName == null) {
        firstName = new SimpleStringProperty();
    }
    return firstName;
}

public final void setLastName(String value) {
    lastNameProperty().set(value);
}

public final String getLastName() {
    return lastNameProperty().get();
}

public StringProperty lastNameProperty() {
    if (lastName == null) {
        lastName = new SimpleStringProperty();
    }
    return lastName;
}
```

```

public final void setMood(MOOD_TYPES value) {
    moodProperty().set(value);
}

public final MOOD_TYPES getMood() {
    return moodProperty().get();
}

public ObjectProperty<MOOD_TYPES> moodProperty() {
    if (mood == null) {
        mood = new SimpleObjectProperty<>(MOOD_TYPES.Happy);
    }
    return mood;
}

public ObservableList<Person> employeesProperty() {
    return employees;
}
}

```

GUI Code

Now that you learned that domain objects (Person) are involved, let's look at the GUI code that will populate and display the data. Listing 6-28 shows the `BossesAndEmployees.java` file containing the main GUI code that will populate a `ListView` control representing the bosses. The rest of the code also populates the `TableView` control on the right side when the user selects a boss.

In Listing 6-28, the GUI code lays out controls programmatically as opposed to FXML via Scene Builder tool. This chapter's code examples primarily take on a programmatic approach, which is often considered bad practice due to the coupling of GUI code and controller code. However, I believe relying on tools to build GUI code will often hide internal mechanics that prevent you from learning how things actually work under the covers. To see how to use FXML views and controller classes, refer to Chapter 5 on layouts and Scene Builder.

Listing 6-28. The Main GUI Code That Creates a Grid Pane Containing a ListView and a TableView Control for Bosses and Employees

```

package com.jfxbe;

import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.geometry.HPos;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.control.cell.ComboBoxTableCell;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.control.cell.TextFieldTableCell;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.paint.Color;

```

```
import javafx.stage.Stage;
import static com.jfxbe.Person.MOOD_TYPES;
import static com.jfxbe.Person.MOOD_TYPES.*;

/**
 * A JavaFX example application to demonstrate
 * observable lists used in ListViews, and TableViews.
 * Also this demos the use of domain objects with
 * properties as attributes.
 *
 * Bosses and Employees
 * @author cdea
 */
public class BossesAndEmployees extends Application {
    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Bosses and Employees: Working with Tables");
        BorderPane root = new BorderPane();
        Scene scene = new Scene(root, 630, 250, Color.WHITE);

        // create a grid pane
        GridPane gridpane = new GridPane();
        gridpane.setPadding(new Insets(20));
        gridpane.setHgap(10);
        gridpane.setVgap(10);
        root.setCenter(gridpane);

        // candidates label
        Label candidatesLbl = new Label("Boss");
        GridPane.setAlignment(candidatesLbl, HPos.CENTER);
        gridpane.add(candidatesLbl, 0, 0);

        // List of bosses
        ObservableList<Person> bosses = getPeople();
        final ListView<Person> leaderListView = new ListView<>(bosses);
        leaderListView.setPrefWidth(150);
        leaderListView.setMinWidth(200);
        leaderListView.setMaxWidth(200);
        leaderListView.setPrefHeight(Integer.MAX_VALUE);

        // display first and last name with tooltip using alias
        leaderListView.setCellFactory(listView -> {
            Tooltip tooltip = new Tooltip();
            ListCell<Person> cell = new ListCell<Person>() {
                @Override
                public void updateItem(Person item, boolean empty) {
                    super.updateItem(item, empty);
                    if (item != null) {
                        setText(item.getFirstName() + " " + item.getLastName());
                        tooltip.setText(item.getAliasName());
                    }
                }
            };
            return cell;
        });
    }
}
```

```

        setTooltip(tooltip);
    }
}
}; // ListCell
return cell;
}); // setCellFactory

gridpane.add(leaderListView, 0, 1);

Label emplLbl = new Label("Employees");
gridpane.add(emplLbl, 2, 0);
GridPane.setHalignment(emplLbl, HPos.CENTER);

TableView<Person> employeeTableView = new TableView<>();
employeeTableView.setEditable(true);
employeeTableView.setPrefWidth(Integer.MAX_VALUE);

ObservableList<Person> teamMembers = FXCollections.observableArrayList();
employeeTableView.setItems(teamMembers);

TableColumn<Person, String> aliasNameCol = new TableColumn<>("Alias");
aliasNameCol.setCellValueFactory(new PropertyValueFactory<>("aliasName"));
aliasNameCol.setCellFactory(TextFieldTableCell.forTableColumn());

TableColumn<Person, String> firstNameCol = new TableColumn<>("First Name");
firstNameCol.setCellValueFactory(new PropertyValueFactory<>("firstName"));
firstNameCol.setCellFactory(TextFieldTableCell.forTableColumn());

TableColumn<Person, String> lastNameCol = new TableColumn<>("Last Name");
lastNameCol.setCellValueFactory(new PropertyValueFactory<>("lastName"));
lastNameCol.setCellFactory(TextFieldTableCell.forTableColumn());

TableColumn<Person, MOOD_TYPES> moodCol = new TableColumn<>("Mood");
moodCol.setCellValueFactory(new PropertyValueFactory<>("mood"));
ObservableList<MOOD_TYPES> moods = FXCollections.observableArrayList(
MOOD_TYPES.values());
moodCol.setCellFactory(ComboBoxTableCell.forTableColumn(moods));
moodCol.setPrefWidth(100);
employeeTableView.getColumns().add(aliasNameCol);
employeeTableView.getColumns().add(firstNameCol);
employeeTableView.getColumns().add(lastNameCol);
employeeTableView.getColumns().add(moodCol);
gridpane.add(employeeTableView, 2, 1);

// selection listening
leaderListView.getSelectionModel()
    .selectedItemProperty()
    .addListener((observable, oldValue, newValue) -> {
if (observable != null && observable.getValue() != null) {
    teamMembers.clear();
}
});

```

```

        teamMembers.addAll(observable.getValue().employeesProperty());
    });
}

primaryStage.setScene(scene);
primaryStage.show();
}

private ObservableList<Person> getPeople() {
    ObservableList<Person> people = FXCollections.<Person>observableArrayList();

    Person docX = new Person("Professor X", "Charles", "Xavier", Positive);
    docX.employeesProperty().add(new Person("Wolverine", "James", "Howlett", Angry));
    docX.employeesProperty().add(new Person("Cyclops", "Scott", "Summers", Happy));
    docX.employeesProperty().add(new Person("Storm", "Ororo", "Munroe", Positive));

    Person magneto = new Person("Magneto", "Max", "Eisenhardt", Sad);
    magneto.employeesProperty().add(new Person("Juggernaut", "Cain", "Marko", Angry));
    magneto.employeesProperty().add(new Person("Mystique", "Raven", "Darkhölme", Sad));
    magneto.employeesProperty().add(new Person("Sabretooth", "Victor", "Creed", Angry));

    Person biker = new Person("Mountain Biker", "Jonathan", "Gennick", Positive);
    biker.employeesProperty().add(new Person("MkHeck", "Mark", "Heckler", Happy));
    biker.employeesProperty().add(new Person("Hansolo", "Gerrit", "Grunwald", Positive));
    biker.employeesProperty().add(new Person("Doc", "José", "Pereña", Happy));
    biker.employeesProperty().add(new Person("Cosmonaut", "Sean", "Phillips", Positive));
    biker.employeesProperty().add(new Person("CarlFX", "Carl", "Dea", Happy));

    people.add(docX);
    people.add(magneto);
    people.add(biker);

    return people;
}

public static void main(String[] args) {
    launch(args);
}
}
```

How It Works

For the fun of it, I've created a simple GUI application that displays bosses and their employees. Notice on the left side of Figure 6-19 a list of important people (bosses) and on the right are employees. When you click to select a boss, their employees will be shown in the TableView control. You'll also notice the tooltip popup when the mouse cursor is hovered over the selected boss.

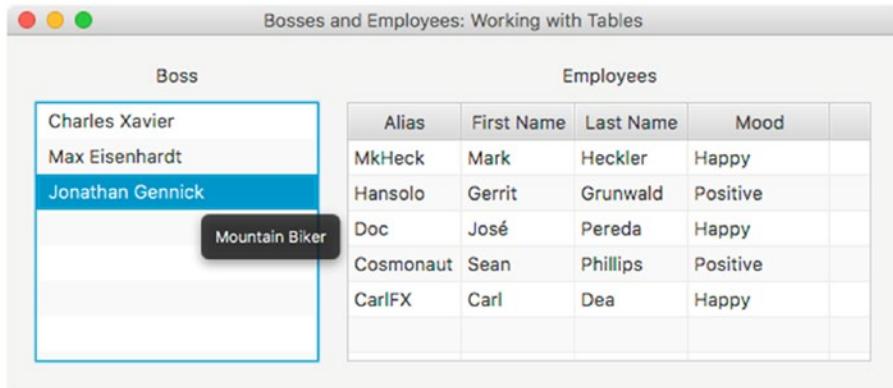


Figure 6-19. The application Bosses and Employees demonstrates the ListView and TableView controls. Note the tooltip displayed when hovering over a boss in the list view.

Listing 6-28 begins by setting up the scene with a primary node as a `GridPane` that will lay out and position the `ListView` and `TableView` controls.

Before discussing the `TableView` control, you need to understand the `ListView` control's responsibility of updating the `TableView`. As discussed earlier about the observable list shown in Listing 6-28, the code populates an `ObservableList` containing all the bosses via the private `getPeople()` method to be passed to the `ListView` control's constructor. The following code is reshown from Listing 6-28. It shows the construction of a `ListView` with an observable list containing domain objects.

```
// List of leaders  
ObservableList<Person> leaders = getPeople();  
ListView<Person> leaderListView = new ListView<>(leaders);
```

Next, the code instantiates a cell factory via the `setCellFactory()` method to properly display the person's name in the `ListView` control's cell. Because each item isn't a string but a `Person` object, the `ListView` does not know how to render each row (cell) in the `ListView` control. To inform the `ListView` which properties to use from the `Person` object, the code simply creates a `javafx.util.Callback` generic object by specifying the `ListView<Person>` and a `ListCell<Person>` data type.

```
// display first and last name with tooltip using alias
leaderListView.setCellFactory(new Callback<ListView<Person>, ListCell<Person>>() {
    @Override
    public ListCell<Person> call(ListView<Person> param) {
        Label leadLbl = new Label();
        Tooltip tooltip = new Tooltip();
        ListCell<Person> cell = new ListCell<Person>() {
            @Override
            public void updateItem(Person item, boolean empty) {
                super.updateItem(item, empty);
                if (item != null) {
                    leadLbl.setText(item.getAliasName());
                    setText(item.getFirstName() + " " + item.getLastName());
                    tooltip.setText(item.getAliasName());
                    setTooltip(tooltip);
                }
            }
        };
        cell.setGraphic(leadLbl);
        cell.setOnMouseClicked(event -> tooltip.show(cell, event.getSceneX(), event.getSceneY()));
        return cell;
    }
});
```

```
        }; // ListCell
    return cell;
}); // setCellFactory
```

You are probably pretty intimidated by the complex anonymous inner class creation of a `Callback` interface. Well, thank goodness there are lambdas in Java 8, because the boilerplate code can be reduced and rewritten as shown in Listing 6-29.

Listing 6-29. The Call Method Displays the Boss's Name in a ListView Control Along with a Tooltip Popup Showing the Alias Name

```
leaderListView.setCellFactory(listView -> {
    Label leadLbl = new Label();
    Tooltip tooltip = new Tooltip();
    ListCell<Person> cell = new ListCell<Person>() {
        @Override
        public void updateItem(Person item, boolean empty) {
            super.updateItem(item, empty);
            if (item != null) {
                leadLbl.setText(item.getAliasName());
                setText(item.getFirstName() + " " + item.getLastName());
                tooltip.setText(item.getAliasName());
                setTooltip(tooltip);
            }
        }
    }; // ListCell
    return cell;
}); // setCellFactory
```

Within the `call()` method shown in Listing 6-28 is the variable `cell` of type `ListCell<Person>`, in which the code creates an anonymous inner class. The inner class must implement the `updateItem()` method. The job of the `updateItem()` method is to obtain each person's information, which then updates a `Label` control (`leadLbl`). The label is displayed for each cell row in the `ListView` control. The last thing the `updateItem()` method does is add a tooltip popup (`Tooltip`) to each cell row. The tooltip will pop up when the cursor hovers over the cell (`boss`) within the `ListView`. After updating the cell, the `cell` variable is returned.

Finally, the code creates a `TableView` control to display the employees based on the selected boss from the `ListView` control. When creating a `TableView` control, the code first creates the *column* headers shown in Listing 6-28.

The creates the `FirstName` table column by setting the property with the JavaFX bean property name of the domain object.

```
TableColumn<Person, String> firstNameCol = new TableColumn<>("First Name");  
firstNameCol.setCellValueFactory(new PropertyValueFactory<>("firstName"));
```

Once you have created a column, you'll notice the `setCellValueFactory()` method, which is responsible for calling the `Person` bean's property. When the list of employees is placed into the `TableView`, it will know how to pull the properties to be placed in each cell column in the table. Based on the JavaBean convention, a new `PropertyValueFactory` instance is created with the name of the bean attribute (`firstName`).

After the columns are set up, the code will set up the cell factories for the `TableView` columns to allow the users to edit cells, which also will update the employee's (`Person`) field. The code will make the `Alias` column into an editable text field and the `Mood` column into an editable combo box.

Last is the implementation of the selection listener on the `ListView`. In the code, you'll notice the `getSelectionModel().selectedItemProperty()` method (shown at the end of Listing 6-28), which allows a listener to be added. The code simply creates and adds a `ChangeListener` for selection events. When a user selects a boss, the `TableView` is cleared and populated with the employees of the selected boss. Actually, it is the magic of the `ObservableList` that notifies the `TableView` of changes. The code that clears and populates the `TableView` is reshown from Listing 6-28 here:

```
teamMembers.clear();
teamMembers.addAll(observable.getValue().employeesProperty());
```

Generating a Background Process

Typically, in desktop applications that have long running processes, visual feedback should indicate that something is happening or suggest that the user be patient. One of the main pitfalls of GUI development is the difficulty of knowing when and how to delegate work to worker threads. Trying to avoid these pitfalls, GUI developers are constantly reminded of thread safety, especially when it comes to blocking the UI thread during heavy loads. JavaFX provides UI controls that help solve this issue by off-loading work to a different thread (`Task`) while providing feedback or progress indicators of the work that is being done.

Creating a Background Task

The JavaFX API can execute work on threads in the background, called tasks (`javafx.concurrent.Task`). These tasks are one-time use that are executed on a thread. The `Task` class is a subclass of Java concurrency's `Runnable` and `Future` interfaces.

What's nice about Java futures is the ability to provide asynchronous behavior when executing code. In other words, when you execute the task, it doesn't block. When using an `ExecutorService`'s `submit()` method, a future object is returned to the caller. The future object can be checked periodically to determine if the task is finished or if an error has occurred. Listing 6-30 shows a simple thread pool created to execute a task.

Listing 6-30. A Thread Pool Created to Execute a Worker Task

```
Task worker = new Task<Boolean>() {
    @Override
    protected Boolean call() throws Exception {
        for (int i=0; i<100; i++) {
            // some heavy work (not on the UI thread)
            updateMessage("progress text");
            updateProgress(i, 100);
        }
        return true;
    }
};

ExecutorService threadPool = threadPool = Executors.newFixedThreadPool(2);
Future future = threadPool.submit(worker);
```

Did you notice the `call()` method in Listing 6-30 containing two methods—`updateMessage()` and `updateProgress()`? These methods assist by updating the status or progress information for the currently running task. Next, you will see an example application that will simulate a dialog window showing progress of files being copied.

Example: File Copy Progress Dialog (BackgroundProcesses)

To demonstrate a GUI application generating a background process while also providing feedback to the user, I've created an example application that simulates a dialog window visualizing the progress of many files being copied. Figure 6-20 depicts, from left to right, a progress bar and a progress indicator (pie chart). The progress indicator shows a percentage below the indicator icon. While the process is still being performed, the user can click on the Cancel button to cancel or terminate the task.

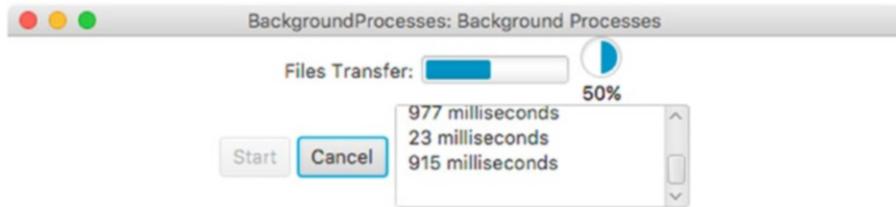


Figure 6-20. A window simulating files being copied using a progress bar and a progress indicator providing user feedback

The following are the main classes used in this example:

- `javafx.scene.control.ProgressBar`
- `javafx.scene.control.ProgressIndicator`
- `javafx.concurrent.Task`

Listing 6-31 shows a code example that uses JavaFX's progress bar and indicator UI controls to provide feedback to the user as files are being copied.

Listing 6-31. An Example Application That Demonstrates Background Processes

```
public class BackgroundProcesses extends Application {

    static Task<Boolean> copyWorker;
    final int numFiles = 30;
    private ExecutorService threadPool;
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }
}
```

```
@Override
public void init() throws Exception {
    super.init();
    threadPool = Executors.newFixedThreadPool(1);
}

@Override
public void stop() throws Exception {
    super.stop();
    threadPool.shutdown();
}

@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("BackgroundProcesses: Background Processes");
    Group root = new Group();
    Scene scene = new Scene(root, 330, 120, Color.WHITE);

    BorderPane mainPane = new BorderPane();
    mainPane.setLayoutXProperty()
        .bind(scene.widthProperty()
            .subtract(mainPane.widthProperty())
            .divide(2)));
    root.getChildren().add(mainPane);

    Label label = new Label("Files Transfer:");
    ProgressBar progressBar = new ProgressBar(0);
    ProgressIndicator progressIndicator = new ProgressIndicator(0);

    HBox hb = new HBox();
    hb.setSpacing(5);
    hb.setAlignment(Pos.CENTER);
    hb.getChildren().addAll(label, progressBar, progressIndicator);
    mainPane.setTop(hb);

    Button startButton = new Button("Start");
    Button cancelButton = new Button("Cancel");
    TextArea textArea = new TextArea();
    textArea.setEditable(false);
    textArea.setPrefSize(200, 70);

    HBox hb2 = new HBox();
    hb2.setSpacing(5);
    hb2.setAlignment(Pos.CENTER);
    hb2.getChildren().addAll(startButton, cancelButton, textArea);
    mainPane.setBottom(hb2);

    // wire up start button
    startButton.setOnAction((ActionEvent event) -> {
        startButton.setDisable(true);
        progressBar.setProgress(0);
```

```

progressIndicator.setProgress(0);
textArea.setText("");
cancelButton.setDisable(false);
copyWorker = createWorker(numFiles);

// wire up progress bar
progressBar.progressProperty().unbind();
progressBar.progressProperty().bind(copyWorker.progressProperty());
progressIndicator.progressProperty().unbind();
progressIndicator.progressProperty().bind(copyWorker.progressProperty());

// append to text area box
copyWorker.messageProperty()
    .addListener((observable, oldValue, newValue) ->
    textArea.appendText(newValue + "\n")
);

threadPool.submit(copyWorker);
});

// cancel button will kill worker and reset.
cancelButton.setOnAction((ActionEvent event) -> {
    startButton.setDisable(false);
    cancelButton.setDisable(true);
    copyWorker.cancel(true);

    // reset
    progressBar.progressProperty().unbind();
    progressBar.setProgress(0);
    progressIndicator.progressProperty().unbind();
    progressIndicator.setProgress(0);
    textArea.appendText("File transfer was cancelled.");
});
cancelButton.setDisable(true);

primaryStage.setScene(scene);
primaryStage.show();
}

private Task<Boolean> createWorker(final int numFiles) {
    return new Task<Boolean>() {

        @Override
        protected Boolean call() throws Exception {
            for (int i = 0; i < numFiles; i++) {
                long elapsedTime = System.currentTimeMillis();
                copyFile("some file", "some dest file");
                elapsedTime = System.currentTimeMillis() - elapsedTime;
                String status = elapsedTime + " milliseconds";
            }
        }
    }
}

```

```

        // queue up status
        updateMessage(status);
        updateProgress(i + 1, numFiles); // (progress, max)
    }
    return true;
}
};

private void copyFile(String src, String dest) throws InterruptedException {
    // simulate a long time
    Random rnd = new Random(System.currentTimeMillis());
    long millis = rnd.nextInt(1000);
    Thread.sleep(millis);
}

}

```

How It Works

The code in Listing 6-31 begins by creating not one but two progress controls to show the user the work being done. Next, the code creates a worker thread via the private `createWorker()` method. The `createWorker()` convenience method instantiates and returns a `javafx.concurrent.Task` object, which is similar to the Java Swing's `SwingWorker` class. Unlike the `SwingWorker` class, the `Task` object is greatly simplified and easier to use, thanks to its event-driven design. An example is a `Task` object's ability to update progress asynchronously.

When creating an instance of a `Task` object, the code implements the `call()` method to do work in the background. As the work is being done, you will often want to queue up intermediate results such as progress or text information to provide feedback to the user. To queue information, the API allows you to call the `updateProgress()` or `updateMessage()` methods. These methods update information in a thread-safe way so that the observer of the progress properties can update the GUI safely without blocking the GUI thread. Listing 6-32 queues up messages and shows the progress.

Listing 6-32. The Text Messages and Progress Updated Without Blocking the Progress UI Controls

```

// queue up status
updateMessage(status);
updateProgress(i + 1, numFiles); // (progress, max progress)

```

After creating a worker task, the code unbinds any old tasks bound to the progress controls. Once the progress controls are unbound, the code then binds the progress controls to a newly created `Task` object named `copyWorker`. Listing 6-33 rebinds a new `Task` object to the progress UI controls.

Listing 6-33. After a Task Is Used, the Progress Controls Will Need to Rebind to a New Task

```

// wire up progress bar
progressBar.progressProperty().unbind();
progressBar.progressProperty().bind(copyWorker.progressProperty());

progressIndicator.progressProperty().unbind();
progressIndicator.progressProperty().bind(copyWorker.progressProperty());

```

Moving on, notice that the code implementing a `ChangeListener` to append the queued results into the `TextArea` control. Another remarkable thing about JavaFX properties is that you can attach many listeners similar to Java Swing components. Finally, the worker and controls are all wired up to spawn a thread to go off in the background. The following code line shows the launching of a Task worker.

```
new Thread(copyWorker).start();
```

This code finishes by wiring up the Cancel button. This button simply calls the `Task` object's `cancel()` method to kill the process. Once the task is cancelled, the progress controls are reset. Once a worker task is cancelled, it cannot be reused. That is why the Start button re-creates a new task. If you want a more robust reusable solution, look at the `javafx.concurrent.Service` class.

Summary

You began this chapter with learning about labels and custom fonts, such as the use of the third-party library `FontAwesomeFX`. After learning about labels, you got a chance to delve deep into various button controls with a fun example that animates game sprites.

After learning common button controls, you learned how to create standard menus and menu bars. By creating menus, you were able to learn to create handler code to respond to the selection of menu items. You were then able to map key combinations or keyboard shortcuts to menu options. You learned how to create context menus invoked by right-clicking on the Scene graph. With the many ways to create menus and menu items, you got a chance to run an example program that simulated a security alarm application.

Finally, you were exposed to `ObservableList` collections, which support the display of UI controls such as the `ListView` and `TableView` nodes. By learning about the UI controls `ListView` and `TableView`, you got a chance to use domain objects to populate list cells and table column cells. You also learned how to make a table view cell column editable using a text field and a combo box.

Lastly, you learned about JavaFX's Task API, which enables you to execute code asynchronously while updating the status of the task. You also learned to bind a task to progress type controls `ProgressBar` and `ProgressIndicator`. With `ProgressBars` and `ProgressIndicators`, you were able to run an example that simulates background worker threads copying files and updating the progress.

CHAPTER 7



Graphics

Have you ever heard someone say, “When two worlds collide”? This expression is used when a person from a different background or culture is put in a situation where they are at odds and must face very hard decisions. When building GUI applications needing graphics or animations, we are often on a collision course between business and gaming worlds. In other words, when developing applications, it’s nice to strike a good balance of having UI aspects from business applications and video game-like effects.

In the ever-changing world of rich Internet applications (RIA), you probably have noticed an increased use of animations such as pulsing buttons, transitions, moving backgrounds, and so on. GUI applications can use animations to provide visual cues to let the user know what to do next. With JavaFX, you will be able to have the best of both worlds, creating appealing graphics as well as form-based applications handling business transactions.

In this chapter, you will learn about the basics of JavaFX graphics such as displaying and manipulating images. Also, in this chapter you will learn high-level animation APIs. Fasten your seatbelts; you’ll discover solutions to integrate cool game-like interfaces into everyday applications.

In this chapter, you’ll learn about the following:

- Loading and displaying images
- Background tasks
- Progress indicators
- Altering color settings on images
- Writing bitmapped images to disk
- Dragging and dropping image files
- Animation using key values, keyframes, and timelines
- Animation transition classes
- Sequential and parallel transitions

Working with Images

Yet another powerful capability in JavaFX is the ability to display standard image file formats on the scene graph. In this section, you learn how to load, display, scale, and rotate images. As a bonus, you’ll also learn how to easily alter an image’s color attributes such as its hue, saturation, brightness, and contrast.

After learning about the basics of the JavaFX `Image` and `ImageView` APIs, you will get to explore an example of a Photo Viewer application. This fun application allows you to scale, rotate, and manipulate images.

Loading Images

You are probably aware of the many standard image file formats on the Internet, such as .jpg, .png, .gif, and .bmp. To load standard image file formats, JavaFX provides the `javafx.scene.image.Image` API. The `Image` class has many convenient constructors that facilitate different loading strategies, as shown in the following list:

- `Image(java.io.InputStream inputStream)`
- `Image(java.io.InputStream is, double requestedWidth, double requestedHeight, boolean preserveRatio, boolean smooth)`
- `Image(java.lang.String url)`
- `Image(java.lang.String url, boolean backgroundLoading)`
- `Image(java.lang.String url, double requestedWidth, double requestedHeight, boolean preserveRatio, boolean smooth)`
- `Image(java.lang.String url, double requestedWidth, double requestedHeight, boolean preserveRatio, boolean smooth, boolean backgroundLoading)`

As you can see, there are only a handful of parameters common to all of the constructors that are used in various combinations for different loading scenarios. Table 7-1 briefly describes each parameter.

Table 7-1. Common Parameters When Loading Images of the Class `javafx.scene.image.Image`

Parameter	Data Type	Description
<code>inputStream</code>	<code>java.io.InputStream</code>	An input stream such as a file or network.
<code>url</code>	<code>String</code>	An image's URL location such as a file on the local filesystem or a web server hosting an image file.
<code>backgroundLoading</code>	<code>boolean</code>	Loads the image in the background (off the JavaFX application thread).
<code>requestedWidth</code>	<code>double</code>	Specifies an image's bounding box width.
<code>requestedHeight</code>	<code>double</code>	Specifies an image's bounding box height.
<code>preserveRatio</code>	<code>boolean</code>	To keep an image's aspect ratio within the bounding box.
<code>smooth</code>	<code>boolean</code>	A true indicates the use of a better algorithm to smooth the image, which can be slower; otherwise it will use the lower quality, which will render pixels faster.

To demonstrate loading an image, Listing 7-1 shows three ways of loading images. The code listing demonstrates loading an image from the local filesystem, classpath, and a remote host system such as a web server.

Listing 7-1. Loading Images from the Filesystem, the Classpath, and a Remote Web Server

```
try {
    // The current thread context should not be on the GUI thread.
    // loading an image from a Windows file system
    File file = new File("C:\\\\Users\\\\jdoe\\\\Pictures\\\\myphoto.jpg");
    String localUrl = file.toURI().toURL().toString();
```

```

// don't load in the background (block)
Image localImage = new Image(localUrl, false);

// loading an image from the classpath in a jar file
String urlStr = this.getClass()
    .getClassLoader()
    .getResource("images/myphoto.jpg")
    .toExternalForm();

// load in the background (non-blocking)
Image cpUrl = new Image(urlStr, true);

// loading an image from a webserver
String remoteUrl = "http://mycompany.com/myphoto.jpg";

// load in the background (non-blocking)
Image remoteImage = new Image(remoteUrl, true);

System.out.println(localUrl); // file:/C:/Users/jdoe/Pictures/myphoto.jpg
System.out.println(cpUrl); // jar:file:/myApp.jar!/images/myphoto.jpg
System.out.println(remoteUrl); // http://mycompany.com/myphoto.jpg

// ... rest of the code
} catch (MalformedURLException ex) {
    ex.printStackTrace();
}

```

This listing demonstrates the use of the constructors receiving a URL path to an image and a boolean parameter to perform the loading of the image in a background thread or not. Since the constructor must accept a URL object, the file's path must be converted to the URL object (URL specification).

Listing 7-1 begins by loading an image from the local filesystem using a File object based on a string with the path location to the image. In this scenario I've specified a Microsoft Windows-based absolute file path for an image. Here you will notice the double backslash in the string representing the file path on the Windows OS. The first backslash is to escape the backslash character in Java Strings. The toURL() method will convert a filesystem path into a standard URL formatted string. What's nice is that the method call converts absolute file paths into the standard Uniform Resource Locator specification's format. The string will contain the prefix `file:` representing the protocol and the path information.

Similar to the local filesystem is loading an image file on the Java classpath or resources directory. Assuming an application was built as an executable JAR, the example in Listing 7-1 shows the protocol as `jar:file:` denoting a file inside a JAR archive file. Pay special attention to the code that uses the getClassLoader() method. Some environments such as OSGi platforms will employ separate class loaders in their own contexts (aka bundle). By using the getClassLoader() method, the code can safely obtain resources from the correct class loader context. For example, if a project has a resources directory containing an image file, you can load a resource based on the classpath as shown:

```

src/
+--main/
+---java/
+----com/
+-----mycompany/
+-----MyClass.java
+---resources/

```

```
+-----com/
+-----mycompany/
+-----my_image.jpg

String urlStr = MyClass.class
    .getClassLoader()
    .getResource("com/mycompany/my_image.jpg")
    .toExternalForm();
```

In this example, you'll notice the resource string `com/mycompany/my_image.jpg` having the path information at the beginning. Sometimes projects will place all image icons inside a directory called `images` or collocated in the package namespace.

Back to the code in Listing 7-1, the last strategy is an example showing the ability to load an image hosted on a remote web server. You can see that the code also uses `http:` as a protocol prefixed to the URL path to demonstrate loading images on a remote host web server.

It's important to keep in mind that the code for Listing 7-1 regarding the statement that makes the blocking call with the boolean value `false` should not be used on the JavaFX application thread. This is due to the possibility of blocking the UI thread, thus making the application appear to freeze up. Depending on size of the image files or network latency, the objective of the UI developer (that's you!) is to not block the UI thread (JavaFX application thread) by loading images on a background thread. After loading an image, the image can now be rendered on the JavaFX application thread (`ImageView` node). If you pass in the Boolean value of `true`, the `ImageView` node will appear blank until the background thread has completed loading the `Image` object.

Now that you know how to load images, you will learn how to display them using the JavaFX `ImageView` node.

Displaying Images

After an `Image` object is loaded, it needs to be passed to an `ImageView` object to be displayed on the JavaFX scene graph. A `javafx.scene.image.ImageView` node is simply a wrapper object that references an `Image` object, which was discussed earlier. Because the `ImageView` object is a JavaFX Node object, you will be able to apply effects and perform transforms.

When an `ImageView` node is used to apply special effects such as blurring, the image's pixel data is not actually manipulated, but copied, to be calculated and displayed onto the `ImageView` node for display. This is quite a powerful technique when there are many `ImageView` objects that all point to a single `Image` object. Listing 7-2 loads an image asynchronously (background loading) to be passed into an `ImageView` constructor.

Listing 7-2. Loading an Image in the Background and Displaying It in an `ImageView` Node

```
Image remoteImage = new Image(remoteUrl, true);
ImageView imageView = new ImageView(remoteImage);
root.getChildren().add(imageView);
```

By default, the `width` and `height` will be the actual dimensions in pixels of the image.

Now that you know how to load and display images programmatically onto the JavaFX Scene graph, let's look at a Photo Viewer example application. In the next section, you see how to implement a Photo Viewer application, which allows images to be loaded, viewed, scaled, and rotated.

A Photo Viewer Example

A practical way to display images in JavaFX is an example of a Photo Viewer application. This application will allow you to drag and drop image files or load from the menu to then be viewed onto the scene graph. Figure 7-1 illustrates the JavaFX Photo Viewer application displaying an image. Figure 7-1 also shows the feature to allow users to modify the current image's color settings.

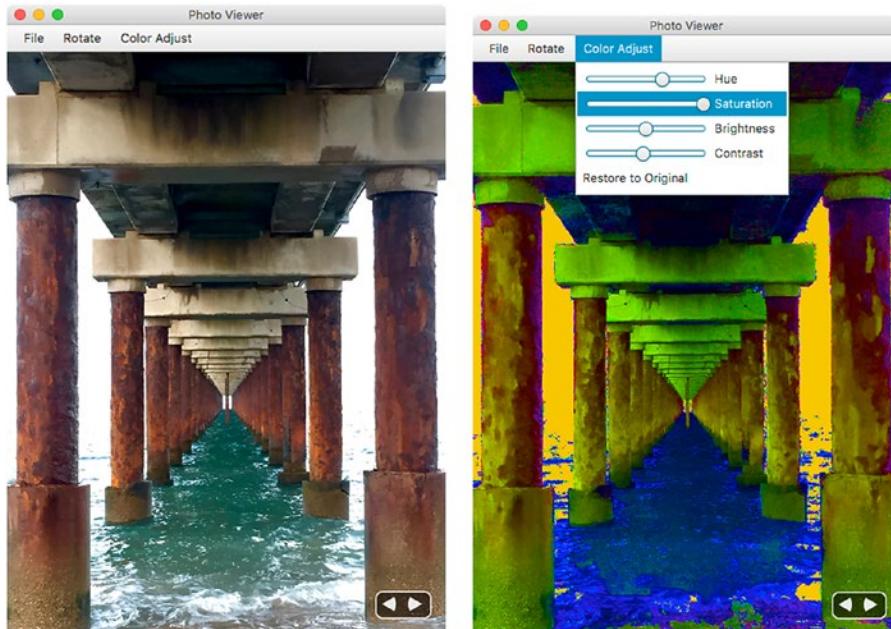


Figure 7-1. The Photo Viewer application displaying the current image. The image on the left is the original photo and the image on the right has color adjustments.

In this section, you will get exposed to the Photo Viewer's features along with instructions on how to use the application. Once you have learned the features and instructions, the section discusses a UML class diagram and a brief description of the source code files. Lastly, you'll get a detailed walkthrough of the Photo Viewer application's source code. The code walkthrough will consist of the application's GUI code, application logic, and CSS styling.

Before looking at the source code of the JavaFX Photo Viewer application, the following section briefly explains the features along with the instructions on how to use it.

Features/Instructions

A Photo Viewer application's features are the following:

- *Loads images:* Loads images having the following file formats: .jpg, .gif, .png, and .bmp. A user can load images by using the Open menu option or by dragging and dropping images from the filesystem into the application area. Figure 7-2 shows the Open menu item to bring up a file chooser.

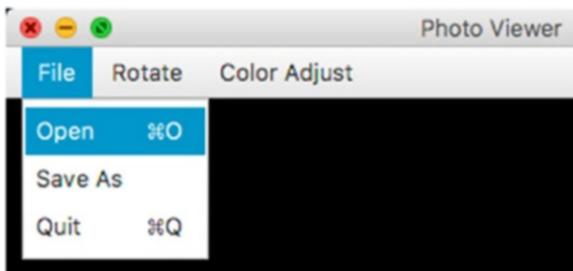


Figure 7-2. Open file menu to present a file chooser to load an image from the filesystem

- *Progress Indicator:* Displays a progress indicator when loading of an image. Figure 7-3 shows a progress indicator when the application is loading images.



Figure 7-3. Progress indicator is displayed to show the application is currently loading a file

- Rotate image: Rotate the image by 90 degrees clockwise or counter-clockwise. Figure 7-4 depicts the menu options to rotate the image by 90 degrees. There are also keyboard shortcuts to rotate the image (the ▶ left/right arrow keys on MacOS and Ctrl left/right arrow keys on Windows and Linux OS).

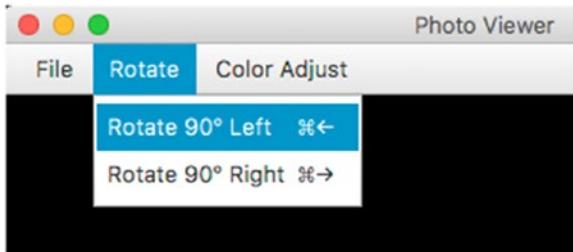


Figure 7-4. Menu options to rotate the image by 90 degrees in clockwise or counter-clockwise direction

- *Resize image:* Resizes the image while preserving the aspect ratio. To resize the current image, use the mouse pointer to expand the application window's height or width.

- *Previous/Next Image buttons:* Ability to page through images using custom button controls. This allows you to see the previous and next image in the list, as shown in Figure 7-5. Also, there are keyboard equivalents by using the arrow keys (left and right) to see the previous and next image, respectively.



Figure 7-5. The custom UI control `ImageViewButtons` is responsible for viewing the previous and next image in the Photo Viewer application. The custom control source code is the `ImageViewButtons.java` file.

- *Adjust color attributes:* The ability to adjust colors of the currently viewed image by changing the hue, saturation, brightness, and contrast. You use the menu options under Color Adjust, as shown in Figure 7-6. Also, you can restore the image to the original color settings.

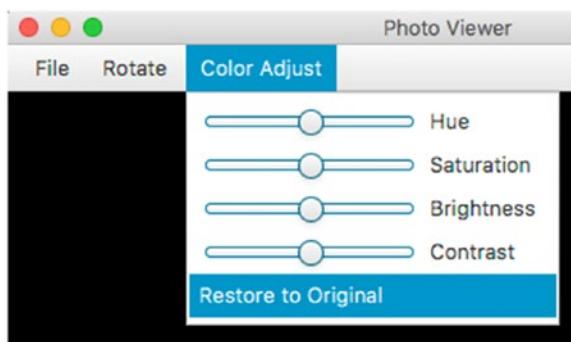


Figure 7-6. The Color Adjust menu options allow you to change the currently displayed image's color attributes. The last option that enables you to restore the image to the original color settings.

- *Save As:* The Save As image menu option allows you to save the currently displayed image with a different filename or overwrite the image it was loaded. The Save As option is on the File menu. The image will be written as bitmap graphics file with the PNG format.

UML: Class Diagram

As a design and class structure, the Photo Viewer application consists of three classes shown in the UML class diagram in Figure 7-7. The three UML classes are `PhotoViewer`, `ImageViewButtons`, and `ImageInfo`. The main JavaFX application class `PhotoViewer` has a one-to-one relationship to an `ImageViewButtons` instance. The `ImageViewButtons` class is a simple custom control that manages zero-to-many `ImageInfo` objects. Basically, the `ImageViewButtons` instance is a UI control that has buttons to view the previous and next image. Each image loaded with effects applied to the current image view will have an associated `ImageInfo` object.

Photo Viewer Class Diagrams

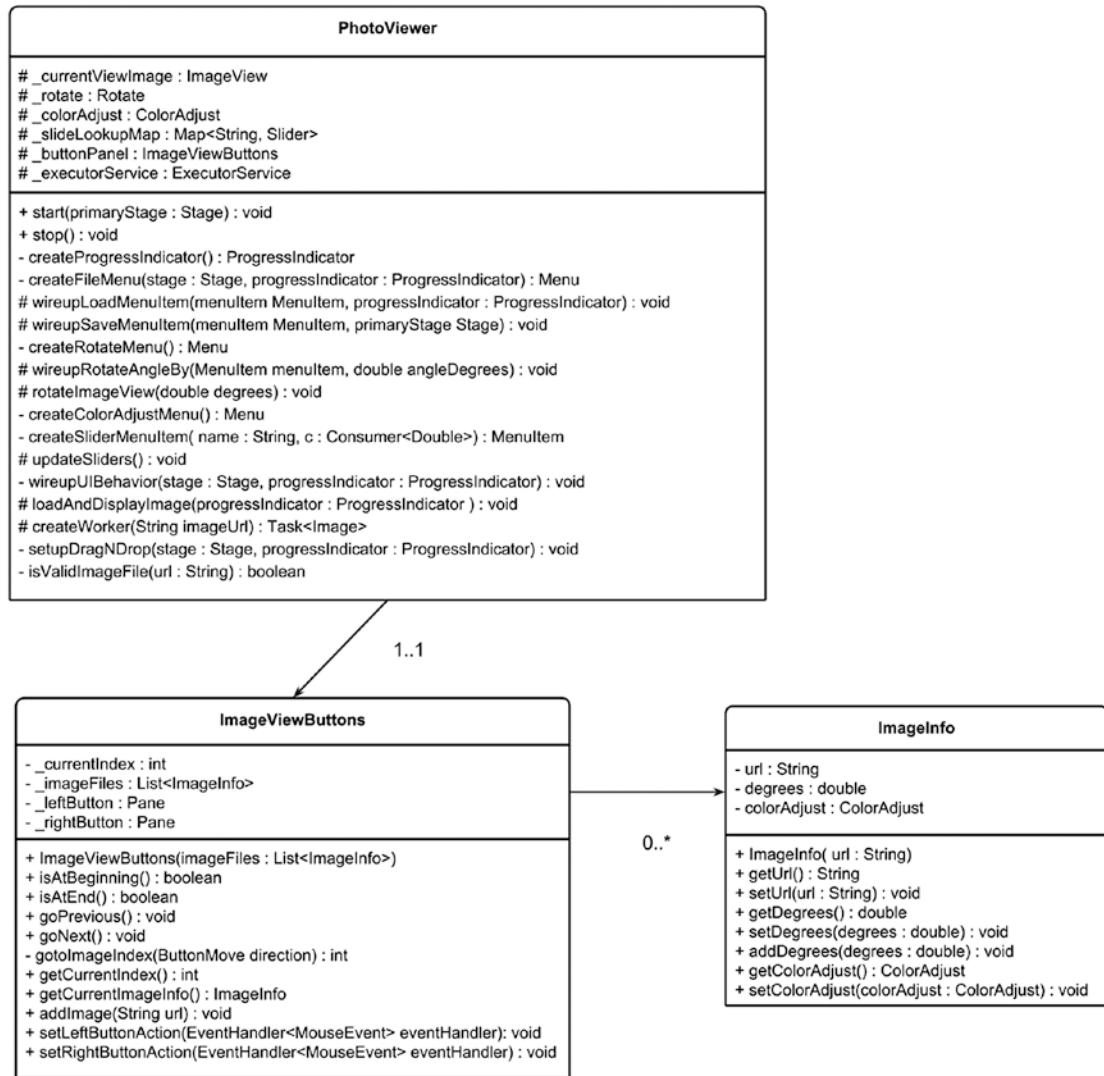


Figure 7-7. Class diagram of the Photo Viewer application. A *PhotoViewer* has an *ImageViewButtons* instance. An *ImageViewButtons* instance has zero-to-many *ImageInfo* objects.

File Descriptions

Before going though a detailed walkthrough of the source code, the next section briefly describes the four source code files that comprise the Photo Viewer example application. The source code of the Photo Viewer example application consists of the following three Java class files and one [JavaFX CSS](#) styling file:

- `PhotoViewer.java`
- `ImageViewButtons.java`

- `ImageInfo.java`
- `photo-viewer.css`

`PhotoViewer.java` is the main JavaFX application that gets launched. Like all examples in this book, you will see a `start()` method that receives a stage (`java.stage.Stage`) instance from the JavaFX application's lifecycle. This is where the code begins to create UI elements that are placed into the JavaFX scene graph.

`ImageViewButtons.java` is a simple custom control that manages a list of string URLs mapped to image files. An instance of `ImageViewButtons` is a custom UI control, as shown earlier in Figure 7-5, depicting the left and right arrow buttons that allows the users to see the previous image and next image.

`ImageInfo.java` is a plain old Java object (POJO) containing an image's URL location, color adjustments, and rotation information. As each image is loaded it will have an associated `ImageInfo` instance. Since the application has only one `ImageView` node that is responsible for displaying a single image at a time, each image has a referenced instance of an `ImageInfo` that stores settings such as the current color adjustments and rotation information. As each image is loaded with the appropriate `ImageInfo` object, that then gets applied to the `ImageView` node.

The `photo-viewer.css` file is a JavaFX CSS styling that is applied to the various UI elements in the Photo Viewer application. Primarily, the CSS file has CSS selectors to colorize buttons and shapes. This file resides in the project's resources directory.

Source Code

In previous editions of this book, the source code example of the Photo Viewer application consisted of only one Java class file. As a single file the only advantage is the code is in one place. However, being one Java class file, there are many disadvantages. One disadvantage that comes to mind is readability, so now the one large file is broken out in three separate Java files and one JavaFX CSS file.

Another notable disadvantage of having things in one Java file is how inflexible the code actually is, especially when adding new features to the application. In fact, at the end of this chapter, you will learn about a new feature of the Photo Viewer application by simply extending the existing application and overriding a method. This chapter consists of multiple code listings that are broken out in code sections. Each Java class file shows multiple code listings to help describe things in more detail.

PhotoViewer.java

Let's begin with source code by looking at Listing 7-3 of the Photo Viewer application. It contains the state variables and constants. Essentially, this is the top portion of the class and file called `PhotoViewer.java`.

These variables being used globally will help cut down on the number of parameters being passed into private methods. Since the `PhotoViewer` class is the focus of the example, it'll be easier to follow by knowing what are global variables versus what are local ones when looking at the code sections.

Listing 7-3. The PhotoViewer Class' State Variables and Constants Declared

```
/** Standard Logger. */
private final static Logger LOGGER = Logger
    .getLogger(PhotoViewer.class.getName());

/** Current image view display */
protected ImageView currentViewImage;
```

```

/** Rotation of the image view */
protected Rotate rotate = new Rotate();

/** Color adjustment */
protected ColorAdjust colorAdjust = new ColorAdjust();

/** A mapping of color adjustment type to a bound slider */
protected Map<String, Slider> sliderLookupMap = new HashMap<>();

/** Custom Button panel to view previous and next images */
protected ImageViewButtons buttonPanel;

/** Single threaded service for loading an image */
protected ExecutorService executorService =
    Executors.newSingleThreadScheduledExecutor();

... // The rest of PhotoViewer.java

```

Listing 7-2 begins by creating a variable named *LOGGER*. It's a standard logger to assist in outputting debug statements into the console. Similar to popular loggers such as Apache log4j, the standard Java logger has various log levels also. To provide an equivalent debug log level like that of log4j, you use the `Level.FINE` as a log level. To see the other log levels, see the Javadoc documentation for details.

Next is a variable named *currentImageView* (`javafx.scene.image.ImageView`) that will represent the main display area of the application to view images. As images are loaded, they will be displayed in the `ImageView` node. The *currentImageView* variable will be created in the `start()` method. Later, you will see how action logic will get attached to the UI elements—but for now we continue discussing the other instance variables.

After declaring the *currentImageView* variable, the *rotate* (`javafx.scene.transform.Rotate`) variable is declared and created. It will be used as a rotate transform for the `ImageView` node. In other words, the *rotate* variable is responsible for rotating the current image in view. As mentioned earlier, each image loaded will have an associated `ImageInfo` object containing its rotation angle and that is applied to the *rotate* variable. Once the current image's degrees are applied, the current image view node will get rotated automatically.

Next, the *colorAdjust* variable is created as a `javafx.scene.effect.ColorAdjust` instance to allow the users to change the image's color attributes of the currently displayed image. Whenever a an image is loaded, an `ImageInfo` instance is created with an instance of a `ColorAdjust` object (via `getColorAdjust()` method). Later, you will see the code for the method `updateSliders()`, which is responsible for updating the menu items that contain sliders and their values. These values bound to each slider will represent the *hue*, *saturation*, *brightness*, and *contrast* levels from the `ImageInfo` object's `colorAdjust` property.

As mentioned about the Color Adjust menu items that contain sliders for adjusting colors of the image, each slider will need to get bound to a color adjustment type. The color adjustment types are hue, saturation, brightness, and contrast. A global variable named *sliderLookupMap* of type `Map<String, Slider>` is a key/value pair that maps a color adjustment type (String) to a JavaFX `Slider` UI control. The lookup map is used later in the code to unbind and bind color adjust properties between the current *colorAdjust* values to each slider control.

One of the last state variables declared is *buttonPanel* of type `ImageViewButtons`. A single instance of a `ImageViewButtons` object is a custom UI control that I created in a class file called `ImageViewButtons.java`. This custom UI control is simply left and right arrow buttons that allow the users to view the previous and next image. Later, you will see the code listing that will dynamically position (layout) the custom control to the bottom right of the application screen when resizing the window.

Lastly, is the *executorService* variable of type `java.util.concurrent.ExecutorService`, which is responsible for running tasks in a background thread. The application's tasks (`javafx.concurrent.Task`) are callbacks that are responsible for loading an image (background thread) and displaying the image once it's been loaded successfully. When displaying the image and applying `ImageInfo` properties, the code that modifies the `ImageView` is executed on the JavaFX Application thread.

After declaring the `PhotoViewer` class' state variables and constants, the code shown in Listing 7-4 shows the main application lifecycle methods implemented. The three methods implemented in Listing 7-4 are `start()`, `stop()`, and `main()`. When the application is launched, the `main()` method runs on the main application thread, which invokes the `launch()` method to begin the JavaFX application lifecycle (the `launch()` method will block).

Listing 7-4. The PhotoViewer Class' start(), stop(), and main() Method Implementations

```
@Override
public void start(Stage primaryStage) {

    primaryStage.setTitle("Photo Viewer");
    BorderPane root = new BorderPane();
    Scene scene = new Scene(root, 551, 400, Color.BLACK);
    scene.getStylesheets()
        .add(getClass()
            .getClassLoader()
            .getResource("photo-viewer.css")
            .toExternalForm());
    primaryStage.setScene(scene);

    // Anchor Pane
    AnchorPane mainContentPane = new AnchorPane();

    // Group is a container to hold the image view
    Group imageGroup = new Group();
    AnchorPane.setTopAnchor(imageGroup, 0.0);
    AnchorPane.setLeftAnchor(imageGroup, 0.0);

    // Current image view
    currentViewImage = createImageView(rotate);
    imageGroup.getChildren().add(currentViewImage);

    // Custom ButtonPanel (Next, Previous)
    List<ImageInfo> IMAGE_FILES = new ArrayList<>();
    buttonPanel = new ImageViewButtons(IMAGE_FILES);

    // Create a progress indicator
    ProgressIndicator progressIndicator = createProgressIndicator();

    // layer items. Items that are last are on top
    mainContentPane.getChildren().addAll(imageGroup,
        buttonPanel, progressIndicator);

    // Create menus File, Rotate, Color adjust menus
    Menu fileMenu = createFileMenu(primaryStage, progressIndicator);
    Menu rotateMenu = createRotateMenu();
    Menu colorAdjustMenu = createColorAdjustMenu();
    MenuBar menuBar = new MenuBar(
        fileMenu, rotateMenu, colorAdjustMenu);
    root.setTop(menuBar);
```

```

// Create the center content of the root pane (Border)
// Make sure the center content is under the menu bar
BorderPane.setAlignment(mainContentPane, Pos.TOP_CENTER);
root.setCenter(mainContentPane);

// When nodes are visible they can be repositioned.
primaryStage.setOnShown( event ->
    wireupUIBehavior(primaryStage, progressIndicator));

primaryStage.show();

}

@Override
public void stop() throws Exception {
    super.stop();
    // Shutdown thread service
    executorService.shutdown();
}

public static void main(String[] args) {
    launch(args);
}

```

When the JavaFX application thread is ready, the `init()` method is called before the `start()` method is called. The `init()` method is invoked after the Application is created. The `init()` method can be overridden to initialize any resources (not run on the JavaFX Application thread). In Listing 7-4, I do not implement the `init()` method, but I wanted to point out the order of methods being called during the application lifecycle. When a JavaFX application exits by using the `Platform.exit()` method, the application's `stop()` method is called last. The `stop()` method gives the application the opportunity to clean up any resources. In Listing 7-4, the `stop()` method calls the `executorService` (`java.util.concurrent.ExecutorService`) variable's `shutdown()` method to gracefully shut down the thread executor service. If the executor service isn't called to shut down, your application will not close properly. After `stop()` executes, the code execution returns to the main thread in the `main()` method after the code line statement invoking the `launch()` shown in Listing 7-4 (last code line).

The main bit of code is located in the `start()` method. The `start()` method begins by creating the root pane as a `BorderPane` to hold menu options at the top (`Pos.TOP`) of the scene and the main UI elements are placed in the center region (`Pos.CENTER`). The code goes on to create the center area with an `AnchorPane` layout to position UI elements relative to the view area. By using a JavaFX Group, any transforms (Rotate) applied to the image view outside of the center area of the `BorderPane` will be pushed down (beneath the menu bar) automatically. The code statements that follow basically call private factory type methods that create and initialize UI elements to be added to the scene graph.

The code goes on to create the center area with an `ImageView` object created using the private method `createImageView()`. The code then creates a single instance of a custom control to manage images with the class `ImageViewButtons`. Next, the code creates the progress indicator via the method `createProgressIndicator()` to let the user know an image is being loaded (busy). Lastly, the code makes calls to create the various menus and menu items such as Open, Save As, etc.

Later, you will see code that will dynamically position the progress indicator and the custom button control (`ImageViewButtons`). But for now, let's take a detailed walkthrough of each of the factory methods that create UI elements.

In Listing 7-5 is a factory function `createImageView()` that is called from the `start()` method responsible for creating an `ImageView` node to be assigned to the global variable `currentImageView`. The created image view node is set to preserve the aspect ratio when the image is being resized via the `setPreserveRatio(true)` method. You'll also notice the `rotate` parameter is passed in as a transform, allowing other code in the application to rotate the image view object.

Listing 7-5. A Factory Type Function Creating an ImageView Having a Rotate Transform

```
private ImageView createImageView(Rotate rotate) {
    ImageView imageView = new ImageView();
    imageView.setPreserveRatio(true);
    imageView.setSmooth(true);
    imageView.getTransforms().addAll(rotate);
    return imageView;
}
```

In Listing 7-6 is a factory function `createProgressIndicator()` that is called from the `start()` method responsible for creating a JavaFX `ProgressIndicator` node to be assigned to the `progressIndicator` instance variable. The created `ProgressIndicator` node using the default constructor will be set to be indeterminate. By default, the code hides the progress indicator UI control via the `setVisible(false)` method. Lastly, the progress indicator's max size is set to 100 by 100 pixels. Later, you'll see the code that dynamically positions the progress indicator in the center of the application window whenever the scene area is resized.

Listing 7-6. A Factory Function Creating a Progress Indicator

```
private ProgressIndicator createProgressIndicator() {
    ProgressIndicator progress = new ProgressIndicator();
    progress.setVisible(false);
    progress.setMaxSize(100d, 100d);
    return progress;
}
```

In Listing 7-7, the `createFileMenu()` method gets called from the `start()` method to create the File menu and other menu items. The code first creates a `Menu` object to contain child menu items such as Open, Save As, and Quit. When creating the Open menu option, you'll notice the string “_Open” having a prefix of an underscore. In order for the menu system to recognize strings with letters prepended with an underscore, you must call the `setMnemonicParse(true)` method.

Listing 7-7. A Factory Function That Creates and Returns a File Menu

```
private Menu createFileMenu(Stage stage,
                           ProgressIndicator progressIndicator) {
    Menu fileMenu = new Menu("File");

    MenuItem loadImagesMenuItem = new MenuItem("_Open");
    loadImagesMenuItem.setMnemonicParsing(true);
    loadImagesMenuItem.setAccelerator(new KeyCodeCombination(KeyCode.O,
        KeyCombination.SHORTCUT_DOWN));

    // file chooser to open a file
    wireupLoadMenuItem(loadImagesMenuItem, stage, progressIndicator);

    MenuItem saveAsMenuItem = new MenuItem("Save _As");
    saveAsMenuItem.setMnemonicParsing(true);

    // file chooser to save image as file
    wireupSaveMenuItem(saveAsMenuItem, stage);
```

```

// Quit application
MenuItem exitMenuItem = new MenuItem("_Quit");
exitMenuItem.setMnemonicParsing(true);
exitMenuItem.setAccelerator(new KeyCodeCombination(KeyCode.Q,
    KeyCombination.SHORTCUT_DOWN));

// exiting
exitMenuItem.setOnAction(actionEvent -> Platform.exit());

fileMenu.getItems().addAll(loadImagesMenuItem,
    saveAsMenuItem, exitMenuItem);

return fileMenu;
}

```

Continuing the discussion of Listing 7-7 regarding underscores prepended to menu text, the underscore before the letter O will allow the JavaFX menu system to provide windowed operating systems that use key modifiers such as the Alt key in combination with a letter to invoke actions. Typically, in a Windows environment when a user presses the Alt key the menus will place an underline beneath a letter of the menu text. In addition to using the modifier combination, the menu item can also be assigned a key accelerator such as ⌘ + O combination based on the Apple keyboard. On Windows- or Linux-based OS, the key accelerator (shortcut) is the Ctrl+O key combination.

The rest of the code is the same in terms of the creation of menu items and wiring up handler code, so I won't go on any further.

Next up is code dealing with menu options to load images. During the creation of the file menu from Listing 7-7 the child menu item `loadImagesMenuItem` is created. At the creation of the `loadImagesMenuItem` menu item, it will get the handler code attached via the `wireupLoadMenuItem()` method, as shown in Listing 7-8.

Listing 7-8. Attach Action Code to Open the Image File Menu Option

```

protected void wireupLoadMenuItem(MenuItem menuItem,
                                Stage primaryStage,
                                ProgressIndicator progressIndicator) {
    // A file chooser is launched with a filter based
    // on image file formats
    FileChooser fileChooser = new FileChooser();
    fileChooser.setTitle("View Pictures");
    fileChooser.setInitialDirectory(
        new File(System.getProperty("user.home")));
    fileChooser.getExtensionFilters().addAll(
        new FileChooser.ExtensionFilter("All Images",
            "*.jpg", "*.jpeg", "*.png", "*.bmp", "*.gif"),
        new FileChooser.ExtensionFilter("JPG", "*.jpg"),
        new FileChooser.ExtensionFilter("JPEG", "*.jpeg"),
        new FileChooser.ExtensionFilter("PNG", "*.png"),
        new FileChooser.ExtensionFilter("BMP", "*.bmp"),
        new FileChooser.ExtensionFilter("GIF", "*.gif"));
    );
    menuItem.setOnAction( actionEvt -> {
        List<File> list = fileChooser.showOpenMultipleDialog(primaryStage);
        if (list != null) {

```

```

        for (File file : list) {
            //openFile(file);
            try {
                String url = file.toURI().toURL().toString();
                if (isValidImageFile(url)) {
                    buttonPanel.addImage(url);
                    loadAndDisplayImage(progressIndicator);
                }
            } catch (MalformedURLException e) {
                e.printStackTrace();
            }
        }
    });
}
}

```

Listing 7-8 begins by creating a JavaFX file chooser that defaults to the user's home directory. To initialize the file chooser to the user's home directory, the code uses the system property `user.home`. To see other system properties, visit the Java documentation at the following link:

<https://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html>

Next, you'll see the various file extension filters added to the file chooser's extension filters property. The code will list (filter) the image files with the proper file extensions. Lastly, the code will invoke the `loadAndDisplayImage(progressIndicator)` method to load and display the image.

During the creation of the File menu from the earlier code in Listing 7-7 the child menu item `saveAsMenuItem` is created. After the creation of the `saveAsMenuItem` menu item, it will get action code attached via `wireupSaveMenuItem()`, as shown in Listing 7-9. The code begins by creating a JavaFX file chooser that defaults to the user's home directory. After the file chooser invokes the `showSaveDialog()` method, the application will block until the user has entered a filename to save the image.

Listing 7-9. Attach Action Code to Save Image Menu Option

```

protected void wireupSaveMenuItem(MenuItem menuItem,
                                Stage primaryStage) {
    menuItem.setOnAction( actionEvent -> {
        FileChooser fileChooser = new FileChooser();
        File fileSave = fileChooser.showSaveDialog(primaryStage);
        if (fileSave != null) {

            WritableImage image = currentViewImage.snapshot(
                new SnapshotParameters(), null);

            try {
                ImageIO.write(SwingFXUtils.fromFXImage(image, null),
                            "png", fileSave);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    });
}
}

```

Once the filename is chosen, the variable `fileSave` will not be `null`, which subsequently takes a snapshot of the `ImageView` node. Invoking the `snapshot()` method essentially returns a `WritableImage` instance containing the bitmap graphics in memory. Lastly, the `ImageIO.write()` method is a convenience method that can write bitmapped graphics to a file.

Next is the creation of the Rotate menu and its menu options. Listing 7-10 shows the `createRotateMenu()` method that is called from the `start()` method. The code first creates a `Menu` object to contain child menu items Rotate 90° Left and Rotate 90° Right.

Listing 7-10. Create and Return the Rotate Menu

```
private Menu createRotateMenu() {
    Menu rotateMenu = new Menu("Rotate");
    // Menu item with a keyboard combo to rotate the image
    // left 90 degrees
    MenuItem rotateLeft = new MenuItem("Rotate 90° Left");
    rotateLeft.setAccelerator(new KeyCodeCombination(KeyCode.LEFT,
        KeyCombination.SHORTCUT_DOWN));
    wireupRotateAngleBy(rotateLeft, -90);

    // Menu item with a keyboard combo to rotate the image
    // right 90 degrees
    MenuItem rotateRight = new MenuItem("Rotate 90° Right");
    rotateRight.setAccelerator(new KeyCodeCombination(KeyCode.RIGHT,
        KeyCombination.SHORTCUT_DOWN));

    wireupRotateAngleBy(rotateRight, 90);

    rotateMenu.getItems().addAll(rotateLeft, rotateRight);
    return rotateMenu;
}
```

Listing 7-10 goes on to attach handler code by invoking the `wireupRotateAngleBy()` method. In Listing 7-11, the `wireupRotateAngleBy()` method's parameters takes the target menu item and an angle in degrees. There you'll notice the call to the `wireupRotateAngleBy()` method with a -90 or 90. The negative or positive 90 degrees will assist in rotating the image in the counter-clockwise or clockwise direction.

In Listing 7-11, the method `wireupRotateAngleBy()` creates action code to be attached to a target menu item with a angle in degrees. The handler code is responsible for obtaining the current image to increase or decrease the angle in degrees based on its image info. It then subsequently invokes the `rotateImageView()` method shown in Listing 7-12.

Listing 7-11. Attach Action Code to Rotate and Rotate Menu Items Right or Left

```
protected void wireupRotateAngleBy(MenuItem menuItem, double angleDegrees) {
    // rotate options
    menuItem.setOnAction(actionEvent -> {
        ImageInfo imageInfo = buttonPanel.getCurrentImageInfo();
        imageInfo.addDegrees(angleDegrees);
        rotateImageView(imageInfo.getDegrees());
    });
}
```

This method eventually calls the `rotateImageView()` method shown in Listing 7-12. The `rotateImageView()` method is the actual code that rotates the `ImageView` node (`rotate` variable). There you'll notice the pivot point is determined based on the center point of the image.

Listing 7-12. The `rotateImageView()` Method Rotates the Image View by an Angle in Degrees. It Sets the Pivot Point in the Center of the Image View Node

```
private void rotateImageView(double degrees) {
    rotate.setPivotX(currentViewImage.getFitWidth()/2);
    rotate.setPivotY(currentViewImage.getFitHeight()/2);
    rotate.setAngle(degrees);
}
```

The last menu option to be created is that of the *Color Adjust* menu. Again, following the same pattern as the other factory methods that create and attach handler code, the `createColorAdjustMenu()` method shown in Listing 7-13 builds up the child menu items to contain JavaFX slider controls to adjust the `ImageView`'s color adjust settings. Each of the child menu items is subsequently created by the `createSliderMenuItem()` method, as shown later in Listing 7-14.

Listing 7-13. Factory Method to Create and Return a Menu with Menu Items that Allow Sliders to Adjust Color Attributes to the Current Image View Node

```
private Menu createColorAdjustMenu() {
    Menu colorAdjustMenu = new Menu("Color Adjust");
    Consumer<Double> hueConsumer = (value) ->
        colorAdjust.hueProperty().set(value);
    MenuItem hueMenuItem = createSliderMenuItem("Hue", hueConsumer);

    Consumer<Double> saturationConsumer = (value) ->
        colorAdjust.setSaturation(value);

    MenuItem saturateMenuItem = createSliderMenuItem("Saturation",
        saturationConsumer);

    Consumer<Double> brightnessConsumer = (value) ->
        colorAdjust.setBrightness(value);

    MenuItem brightnessMenuItem = createSliderMenuItem("Brightness",
        brightnessConsumer);

    Consumer<Double> contrastConsumer = (value) ->
        colorAdjust.setContrast(value);

    MenuItem contrastMenuItem = createSliderMenuItem("Contrast",
        contrastConsumer);

    MenuItem resetMenuItem = new MenuItem("Restore to Original");

    resetMenuItem.setOnAction(actionEvent -> {
        colorAdjust.setHue(0);
        colorAdjust.setContrast(0);
        colorAdjust.setBrightness(0);
```

```

        colorAdjust.setSaturation(0);
        updateSliders();
    });

colorAdjustMenu.getItems()
    .addAll(hueMenuItem, saturateMenuItem,
            brightnessMenuItem, contrastMenuItem,
            resetMenuItem);

return colorAdjustMenu;
}

```

Continuing the discussion of Listing 7-13, each child menu item is created by calling the `createSliderMenuItem()` method, as shown in Listing 7-14 below. The method signature has the following parameters passed in: `name` and `c` (`Consumer`). Each of the color adjustment types will have a JavaFX Slider control associated with it. The string name is used in the `sliderLookupMap` as the key representing the color adjustment types when looking up a Slider. The string keys are the following: Hue, Saturation, Brightness, and Contrast.

Listing 7-14. The `createSliderMenuItem()` Method Creates a Menu Item with a Slider Control to Change a Particular Color Adjustment Such as Hue, Saturation, Brightness, and Contrast

```

private MenuItem createSliderMenuItem(String name,
                                     Consumer<Double> c) {

    Slider slider = new Slider(-1, 1, 0);
    sliderLookupMap.put(name, slider);
    slider.valueProperty().addListener(ob ->
        c.accept(slider.getValue()));

    Label label = new Label(name, slider);
    label.setContentDisplay(ContentDisplay.LEFT);
    MenuItem menuItem = new CustomMenuItem(label);
    return menuItem;
}

```

Listing 7-14 shows an `InvalidationListener` added to the `value` property of the slider using the `addListener()` method. The listener will execute the functional interface `c` (`Consumer<Double>`). Whenever you see this programming pattern where the caller is passing in a functional interface as a parameter, it usually means the caller wants a callback type behavior or deferred code to be executed. Often developers will define a block of execution code (a closure) outside of the current function scope. To see many convenient functional interfaces used in Java 8, see the package `java.util.function.*` in the Javadoc documentation.

In Listing 7-14, the method parameter begins with the `String name`. The `name` parameter is used as the text label that displays the color adjust type, such as hue, saturation, brightness, or contrast. The `name` parameter is also the string of the color adjust type that's used as a key to retrieve the Slider control from the `sliderLookupMap`. Lastly, the `c` parameter represents the handler code for each slider that is associated with a color adjust property. The parameter `c` is a `java.util.function.Consumer` functional interface that is used as handler code that gets invoked when the slider's value changes. For example, when the caller creates a slider for the hue property, the caller will pass in a `Consumer` functional interface like the following:

```

Consumer<Double> hueConsumer = (value) -> colorAdjust.hueProperty().set(value);
MenuItem hueMenuItem = createSliderMenuItem("Hue", hueConsumer);

```

The `createSliderMenuItem()` function is generic and it allows the caller to define the behavior when the slider's value property is changing.

Additionally, Listing 7-15 updates the color adjust sliders and their values. Whenever the user views a previous or next image, the code will invoke the `updateSliders()` method, which takes the current image's associated image info (`ImageInfo`) object and repositions the knobs of the color adjust menu item sliders to the new values.

Listing 7-15. The `updateSlider()` Method Is Responsible for Updating the Menu Items Containing Slider Controls to Update Their Values Based on the Currently Viewed Image (`ImageInfo`)

```
protected void updateSliders() {
    sliderLookupMap.forEach( (id, slider) -> {
        switch (id) {
            case "Hue":
                slider.setValue(colorAdjust.getHue());
                break;
            case "Brightness":
                slider.setValue(colorAdjust.getBrightness());
                break;
            case "Saturation":
                slider.setValue(colorAdjust.getSaturation());
                break;
            case "Contrast":
                slider.setValue(colorAdjust.getContrast());
                break;
            default:
                slider.setValue(0);
        }
    });
}
```

Listing 7-16 then takes further steps to attach the handler code to the buttons and includes presentation logic to various UI elements. The additional presentation logic begins with the handler code to recalculate and reposition the custom button panel (`buttonPanel1`) and the progress indicator, respectively. The intent is to have the custom button control appear to float at the bottom-right corner whenever the window is resized.

Listing 7-16. The `wireupUIBehavior()` Method Attaches Handler Code to UI Elements

```
private void wireupUIBehavior(Stage primaryStage,
                             ProgressIndicator progressIndicator) {
    Scene scene = primaryStage.getScene();

    // make the custom button panel float bottom right
    Runnable repositionButtonPanel = () -> {
        // update buttonPanel's x
        buttonPanel.setTranslateX(scene.getWidth() - 75);
        // update buttonPanel's y
        buttonPanel.setTranslateY(scene.getHeight() - 75);
    };

    // make the progress indicator float in the center
    Runnable repositionProgressIndicator = () -> {
```

```

// update progress x
progressIndicator.setTranslateX(
    scene.getWidth()/2 - (progressIndicator.getWidth()/2));
progressIndicator.setTranslateY(
    scene.getHeight()/2 - (progressIndicator.getHeight()/2));
};

// invoking both to repositioning closures.
Runnable repositionCode = () -> {
    repositionButtonPanel.run();
    repositionProgressIndicator.run();
};

// Anytime the window is resized reposition the button panel
scene.widthProperty().addListener(observable ->
    repositionCode.run());
scene.heightProperty().addListener(observable ->
    repositionCode.run());

// Go ahead and reposition now
repositionCode.run();

// resize image view when scene is resized.
currentViewImage.fitWidthProperty()
    .bind(scene.widthProperty());

// view previous image action
Runnable viewPreviousAction = () -> {
    // if no previous image or currently loading.
    if (buttonPanel.isAtBeginning()) return;
    else buttonPanel.goPrevious();
    loadAndDisplayImage(progressIndicator);
};

// attach left button action
buttonPanel.setLeftButtonAction( mouseEvent ->
    viewPreviousAction.run());

// Left arrow key stroke pressed action
scene.addEventHandler(KeyEvent.KEY_PRESSED, keyEvent -> {
    if (keyEvent.getCode() == KeyCode.LEFT
        && !keyEvent.isShortcutDown()) {
        viewPreviousAction.run();
    }
});

// view next image action
Runnable viewNextAction = () -> {
    // if no next image or currently loading.
    if (buttonPanel.isAtEnd()) return;
    else buttonPanel.goNext();
    loadAndDisplayImage(progressIndicator);
};

```

```

// attach right button action
buttonPanel.setRightButtonAction( mouseEvent ->
    viewNextAction.run());

// Right arrow key stroke pressed action
scene.addEventHandler(KeyEvent.KEY_PRESSED, keyEvent -> {
    if (keyEvent.getCode() == KeyCode.RIGHT
        && !keyEvent.isShortcutDown()) {
        viewNextAction.run();
    }
});

// Setup drag and drop file capabilities
setupDragNDrop(primaryStage, progressIndicator);
}

```

The `wireupUIBehavior()` method from Listing 7-16 finishes things off by adding handler code to buttons and keystrokes to view the previous and next image. The very last bit of setup in Listing 7-16 is a call to the `setupDragNDrop()` method, which will be discussed later, during the walkthrough in Listing 7-19.

When the user goes to view the previous or next image, the code to load and display an image is the method `loadAndDisplayImage()` shown in Listing 7-17. Notice that the progress indicator is passed in as a parameter. The `progressIndicator` is hidden by default (its `visible` property is `false`). As the images are in the process of loading, the progress indicator becomes visible. It's then hidden when the loading process is complete.

Listing 7-17. The `loadDisplayImage()` Method Is Responsible for Loading an Image from a URL and Displaying the Image

```

protected void loadAndDisplayImage(ProgressIndicator progressIndicator) {
    if (buttonPanel.getCurrentIndex() < 0) return;

    final ImageInfo imageInfo = buttonPanel.getCurrentImageInfo();

    // show spinner while image is loading
    progressIndicator.setVisible(true);

    Task<Image> loadImage = createWorker(imageInfo.getUrl());

    // after loading has succeeded apply image info
    loadImage.setOnSucceeded(workerStateEvent -> {

        try {
            currentViewImage.setImage(loadImage.get());

            // Rotate image view
            rotateImageView(imageInfo.getDegrees());

            // Apply color adjust
            colorAdjust = imageInfo.getColorAdjust();
            currentViewImage.setEffect(colorAdjust);
        }
    });
}

```

```

        // update the menu items containing slider controls
        updateSliders();

    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    } finally {
        // hide progress indicator
        progressIndicator.setVisible(false);
    }

});

// any failure turn off spinner
loadImage.setOnFailed(workerStateEvent ->
    progressIndicator.setVisible(false));

executorService.submit(loadImage);
}

```

Listing 7-17 calls the `createWorker()` method that returns a Task worker the code then adds handler code that responds when the success event is raised via the `setOnSucceeded()` method. Here, the code will set the image, apply the rotation angle, apply color adjustments to the image view, and update the color adjust sliders in the menu options.

Every time a call is made to the `loadAndDisplayImage()` method, it will create a new `Task<Image>` callback to be run on the executor service (`executorService`). Listing 7-18 shows the creation of a new Task responsible for loading images on a background thread and displaying the image on the JavaFX application thread.

Listing 7-18. Creating a Task Based on a URL to Load and Render an Image

```

protected Task<Image> createWorker(String imageUrl) {
    return new Task<Image>() {
        @Override
        protected Image call() throws Exception {
            // On the worker thread...
            Image image = new Image(imageUrl, false);
            return image;
        }
    };
}

```

As you are almost finished with this detailed walkthrough of the `PhotoViewer.java` source code, the only code functions left are Listings 7-19 and 7-20. Listing 7-19 shows the `setupDragNDrop()` method mentioned earlier that's related to handling files that are dragged and dropped over the application. The `setupDragNDrop()` method begins by adding handler code whenever a `DragOver` event is raised.

When files are dragged and dropped over the surface of the scene, the handler code obtains the `DragBoard` object to determine if there were valid image files before proceeding. Once the list of files is valid, the event is updated with a `TransferMode.LINK`. It shows an icon at the tip of your mouse pointer providing visual feedback on the drag process over the surface. The `setupDragNDrop()` code continues with the handler code that is invoked when a `DragDropped` event is raised.

Listing 7-19. The setupDragNDrop() Method Is Responsible for the Drag and Drop Image File Handling Behavior

```

private void setupDragNDrop(Stage primaryStage,
                           ProgressIndicator progressIndicator) {
    Scene scene = primaryStage.getScene();

    // Dragging over surface
    scene.setOnDragOver((DragEvent event) -> {
        Dragboard db = event.getDragboard();
        if ( db.hasFiles()
            || (db.hasUrl()
                && isValidImageFile(db.getUrl())))
        {
            LOGGER.log(Level.INFO, "url " + db.getUrl());
            event.acceptTransferModes(TransferMode.LINK);
        } else {
            event.consume();
        }
    });

    // Dropping over surface
    scene.setOnDragDropped((DragEvent event) -> {
        Dragboard db = event.getDragboard();
        // image from the local file system.
        if (db.hasFiles() && !db.hasUrl())
        {
            db.getFiles().forEach( file -> {
                try {
                    String url = file.toURI().toURL().toString();
                    if (isValidImageFile(url))
                        buttonPanel.addImage(url);
                }

                } catch (MalformedURLException ex) {
                    ex.printStackTrace();
                }
            });
        } else {
            String url = db.getUrl();
            LOGGER.log(Level.FINE, "dropped url: "+ db.getUrl());
            if (isValidImageFile(url))
                buttonPanel.addImage(url);
        }
    });

    loadAndDisplayImage(progressIndicator);

    event.setDropCompleted(true);
    event.consume();
});
}

```

During the load image process, the files are validated by checking if files have valid image file extensions. To validate the file extension, Listing 7-20 shows the `isValidImageFile()` method. Here, you'll see the use of the stream API and the `anyMatch()` method. It converts the URL string to lowercase and checks the end of the string against the list of image type extensions.

Listing 7-20. The `isValidImageFile()` Returns True if the URL String Has a File Extension of JPG, JPEG, PNG, GIF, or BMP

```
private boolean isValidImageFile(String url) {
    List<String> imgTypes = Arrays.asList(".jpg", ".jpeg",
        ".png", ".gif", ".bmp");
    return imgTypes.stream()
        .anyMatch(t -> url.toLowerCase().endsWith(t));
}
```

ImageViewButtons.java

Listing 7-21 shows a simple custom UI control displaying the previous and next buttons. The custom control also manages the list of images. The state variables that help to do bookkeeping of the image files are `currentIndex`, `imageFiles`, `leftButton`, and `rightButton`. The `currentIndex` variable keeps track of the current index into the `imageFiles` list of a displayed image. The current index is defaulted as -1 to denote that there are no images loaded. An enumeration `ButtonMove` was created to help internally to determine whether either the previous or next button was pressed.

Listing 7-21. The Custom ImageViewButtons Class Is the UI Code Providing Left and Right Buttons to View the Previous and Next Images

```
/**
 * Created by cpdea on 11/12/16.
 */
public class ImageViewButtons extends Pane {
    /** The current index into the IMAGE_FILES list. */
    private int currentIndex = -1;

    /** Enumeration of next and previous button directions */
    public enum ButtonMove {NEXT, PREV}

    /** List of ImageInfo instances. */
    private List<ImageInfo> imageFiles;

    private Pane leftButton;
    private Pane rightButton;

    public ImageViewButtons(List<ImageInfo> imageFiles) {
        imageFiles = imageFiles;

        // create button panel
        Pane buttonStackPane = new StackPane();
        buttonStackPane.getStyleClass().add("button-pane");

        // left arrow button
        leftButton = new Pane();
```

```
Arc leftButtonArc = new Arc(0,12, 15, 15, -30, 60);
leftButton.getChildren().add(leftButtonArc);

leftButtonArc.setType(ArcType.ROUND);
leftButtonArc.getStyleClass().add("left-arrow");

// Right arrow button
rightButton = new Pane();
Arc rightButtonArc = new Arc(15, 12, 15, 15, 180-30, 60);
rightButton.getChildren().add(rightButtonArc);
rightButtonArc.setType(ArcType.ROUND);
rightButtonArc.getStyleClass().add("right-arrow");

HBox buttonHbox = new HBox();
buttonHbox.getStyleClass().add("button-panel");
HBox.setHgrow(leftButton, Priority.ALWAYS);
HBox.setHgrow(rightButton, Priority.ALWAYS);
HBox.setMargin(leftButton, new Insets(0,5,0,5));
HBox.setMargin(rightButton, new Insets(0,5,0,5));
buttonHbox.getChildren().addAll(leftButton, rightButton);

buttonStackPane.getChildren().addAll(buttonHbox);

getChildren().add(buttonStackPane);
}

public boolean isAtBeginning() {
    return currentIndex == 0;
}

public boolean isAtEnd() {
    return currentIndex == imageFiles.size()-1;
}

public void goPrevious() {
    currentIndex = gotoImageIndex(ButtonMove.PREV);
}

public void goNext() {
    currentIndex = gotoImageIndex(ButtonMove.NEXT);
}

private int gotoImageIndex(ButtonMove direction) {
    int size = imageFiles.size();
    if (size == 0) {
        currentIndex = -1;
    } else if (direction == ButtonMove.NEXT
        && size > 1
        && currentIndex < size - 1) {
        currentIndex += 1;
    } else if (direction == ButtonMove.PREV
        && size > 1
        && currentIndex > 0) {
        currentIndex -= 1;
    }
}
```

```

        return currentIndex;
    }
    public int getCurrentIndex() {
        return currentIndex;
    }
    public ImageInfo getCurrentImageInfo() {
        return imageFiles.get(getCurrentIndex());
    }

    /**
     * Adds the URL string representation of the path to the image file.
     * Based on a URL the method will check if it matches supported
     * image format.
     * @param url string representation of the path to the image file.
     */
    public void addImage(String url) {
        currentIndex +=1;
        imageFiles.add(currentIndex, new ImageInfo(url));
    }
    public void setLeftButtonAction(EventHandler<MouseEvent> eventHandler) {
        leftButton.addEventHandler(MouseEvent.MOUSE_PRESSED, eventHandler);
    }
    public void setRightButtonAction(EventHandler<MouseEvent> eventHandler) {
        rightButton.addEventHandler(MouseEvent.MOUSE_PRESSED, eventHandler);
    }
}

```

The custom button panel control code has callback methods to allow users of the API to be able to attach handler code to the previous and next buttons. The methods to attach the handler code are called `setLeftButtonAction()` and `setRightButtonAction()`. For example, the following code snippet allows the user of the API to attach handler code when the button is clicked:

```
setLeftButtonAction((mouseEvent) -> System.out.println("JavaFX 9"));
```

ImageInfo.java

The `ImageInfo` class contains various image display properties such as the URL location of the image file. The `ImageInfo` class, as shown in Listing 7-22, is also responsible for holding the rotation angle (degrees) and the color adjust (`ColorAdjust`) settings made by the user. As each image is viewed using the previous and next buttons, the associated `ImageInfo` instance is applied to the `ImageView` object.

Listing 7-22. The `ImageInfo` Class Is a POJO to Hold the Image's URL, Rotation Angle, and Color Adjust Settings

```

public class ImageInfo {
    private String url;
    private double degrees;
    private ColorAdjust colorAdjust;

```

```

public ImageInfo(String url) {
    this.url = url;
}
public String getUrl() {
    return url;
}

public void setUrl(String url) {
    this.url = url;
}

public double getDegrees() {
    return degrees;
}

public void setDegrees(double degrees) {
    this.degrees = degrees;
}

public void addDegrees(double degrees) {
    setDegrees(this.degrees + degrees);
}

public ColorAdjust getColorAdjust() {
    if (colorAdjust == null) {
        colorAdjust = new ColorAdjust();
    }
    return colorAdjust;
}

public void setColorAdjust(ColorAdjust colorAdjust) {
    this.colorAdjust = colorAdjust;
}
}

```

photo-viewer.css

The JavaFX CSS styling for the Photo Viewer application is the `photo-viewer.css` file, shown in Listing 7-23. If you aren't familiar with JavaFX CSS styling, you can fast-forward to Chapter 15 on custom user interfaces. In short, in Listing 7-23 contains JavaFX CSS styling for the custom control (`ImageViewButtons`) from Listing 7-21. Typically, you can add class selectors to any JavaFX node by invoking the node's `getStyleClass().add(...)` method.

For example, here I've set the custom button's area node to use the class selector "button-pane":

```
buttonStackPane.getStyleClass().add("button-pane")
```

Listing 7-23. The photo-viewer.css File Contains the JavaFX CSS Selectors and Styling Definitions for the Various UI Elements

```

/*
File    : photo-viewer.css
Author  : Carl Dea
*/
.root {
    -fx-background-color: rgba(0, 0, 0, .75);
    -arrow-fill: rgba(255, 255, 255, .90);
    -arrow-fill-hover: rgba(71, 241, 255, .90);
    -blur-effect-hover: dropshadow(gaussian, rgba(255, 255, 255, .90), 10, .3, 0, 0 );
}
.button-pane {
    -fx-min-width: 60;
    -fx-max-width: 60;
    -fx-pref-width: 60;
    -fx-min-height: 30;
    -fx-max-height: 30;
    -fx-pref-height: 30;
    -fx-border-radius: 7.5;
    -fx-background-radius: 9.0;
    -fx-background-color: rgba(0,0,0, .55);
    -fx-border-width: 1.5;
    -fx-border-color: white;
}
.left-arrow {
    -fx-fill: -arrow-fill;
}
.left-arrow:hover {
    -fx-fill: -arrow-fill-hover;
    -fx-effect: -blur-effect-hover;
}
.right-arrow {
    -fx-fill: -arrow-fill;
}
.right-arrow:hover {
    -fx-fill: -arrow-fill-hover;
    -fx-effect: -blur-effect-hover;
}

```

The remainder of Listing 7-23 shows CSS selectors employing pseudo-classes that handle mouse hover (`:hover`) states when the mouse cursor enters or exits a node. When the mouse hovers over the previous and next arrow buttons, a blue glow effect is applied to the button. In Chapter 15, you will get a chance to hear more about class selectors and pseudo-class states.

Well, there you have it, a nice Photo Viewer application capable of scaling, rotating, and changing color settings in an image. Now for even more fun on the topic of graphics, the next section looks at JavaFX animation APIs.

Animation

An *animation* is an illusion of motion when images change over time. Quite similar to a cartoon flipbook, each page represents a frame or picture that will be displayed on a timeline for a period of time (duration). In JavaFX, you will be working with the animation API ([javafx.animation.*](#)). In this section, you learn about key values, keyframes, timelines, and transitions. Finally, to demonstrate the animation APIs, you will explore a slightly modified Photo Viewer application with additional enhancements.

What Are Key Values?

The JavaFX animation API allows you to assemble timed events that can interpolate over property-based values in order to produce animated effects. For example, to produce a fade-out effect, you target a node's opacity property to interpolate its value starting from 1 (fully opaque) to 0 (transparent) over a period of time. Listing 7-24 defines a KeyValue instance that targets a rectangle node's opacity property starting at 1 and ending with the value 0. By default, a KeyValue object will have a linear interpolator.

Listing 7-24. A KeyValue Object to Fade Out by Interpolating the Opacity Property From 1 to 0

```
Rectangle Rectangle rectangle = new Rectangle(0, 0, 50, 50);
KeyValue keyValue = new KeyValue(rectangle.opacityProperty(), 0);
```

The KeyValue object doesn't actually interpolate the value, but simply defines the start and end values of a property to interpolate between. Also, the KeyValue can be defined with different types of interpolators, such as linear, ease in, or ease out. For example, Listing 7-25 defines a key value that will animate a rectangle by moving it from left to right by 100 pixels having an `Interpolator.EASE_OUT` interpolator. Easing out will slow down the movement before the end key value.

Listing 7-25. A KeyValue Object to Move a Rectangle from Left to Right 100 Pixels with an Interpolator that Starts to Slow Down When it Reaches the End Value

```
Rectangle rectangle = new Rectangle(0, 0, 50, 50);
KeyValue keyValue = new KeyValue(rectangle.xProperty(), 100, Interpolator.EASE_OUT);
```

Visit the Javadocs documentation for available interpolators ([javafx.animation.Interpolator](#)). By default, KeyValue constructors that don't specify an interpolator will use a linear interpolation (`Interpolator.LINEAR`), which evenly distributes the values over a period of time.

What Are Keyframes?

When an animation ([javafx.animation.Timeline](#)) occurs, each timed event, called a *keyframe* (a KeyFrame object), is responsible for interpolating key values (KeyValue objects) over a period of time ([javafx.util.Duration](#)). When creating a KeyFrame object, the constructor requires a timed duration to interpolate over key values. KeyFrame constructors all accept one or more key values by using a variable containing an argument list or a collection of KeyValue objects.

To demonstrate moving a rectangle in a diagonal direction from upper left to lower right, Listing 7-26 defines a keyframe that has a duration of 1000 milliseconds and two key values that represent the rectangle's x and y properties.

Listing 7-26. Keyframe Defined for a Rectangle to Move Diagonally from Upper Left to Lower Right Spanning a One-Second Duration

```
Rectangle rectangle = new Rectangle(0, 0, 50, 50);
KeyValue xValue = new KeyValue(rectangle.xProperty(), 100);
KeyValue yValue = new KeyValue(rectangle.yProperty(), 100);
KeyFrame keyFrame = new KeyFrame(Duration.millis(1000), xValue, yValue);
```

This example defines a keyframe to move a rectangle's upper-left corner (0, 0) to point (100, 100) in one second or 1000 milliseconds. Now that you know how to define keyframes, you need to execute the animation by using a `Timeline` instance.

What Is a Timeline?

A *timeline* is one animation sequence consisting of many `KeyFrame` objects. Each `KeyFrame` object is run sequentially. Because a timeline is a subclass of the abstract `javafx.animation.Animation` class, it has standard attributes, such as its cycle count and auto-reverse, that you can set. The cycle count is the number of times you want the timeline to play the animation. If you want the cycle count to play the animation indefinitely, use the value `Timeline.INDEFINITE`. The auto-reverse property is a Boolean flag to indicate that the animation can play the timeline backward (the keyframes in reverse order). By default, the cycle count is set to 1 and auto-reverse is set to false.

To add keyframes to a `Timeline` object, use the `getKeyFrames().addAll()` method. Listing 7-27 demonstrates a timeline playing indefinitely (looping) with auto-reverse set to true to enable the animation sequence to go backward. A `Timeline` instance is created with a cycle count set to run indefinitely and auto-reverse set to true to allow the animation sequence to go backward.

Listing 7-27. A Timeline Instance Repeating an Animation Indefinitely

```
Timeline timeline = new Timeline();
timeline.setCycleCount(Timeline.INDEFINITE);
timeline.setAutoReverse(true);
timeline.getKeyFrames().addAll(keyFrame1, keyFrame2);
timeline.play();
```

With this knowledge of timelines, you can now animate any scene graph node in JavaFX. Although you have the ability to create timelines in a low-level way, this technique can become very cumbersome for simple animations. You are probably wondering whether there are easier ways to express common animation effects such as *fading*, *scaling*, and *translating*. Good news! JavaFX has stock (built-in) transition animations (extending the `Transition` class), which are convenience classes to perform common animated effects.

JavaFX Transition Classes

The JavaFX API supports common animations called *transition classes*. Transition classes inherit the `javafx.animation.Animation` class similar to a `Timeline` discussed earlier. Transition classes are simpler ways to create common animations that get applied to JavaFX Node objects.

Here are some of the most common transition animation classes, followed by a brief description of each:

- `javafx.animation.FadeTransition`
- `javafx.animation.PathTransition`
- `javafx.animation.ScaleTransition`
- `javafx.animation.TranslateTransition`

The fade transition (`FadeTransition`) targets the node's `opacity` property for a fading animation effect. The path transition (`PathTransition`) enables a node to follow a generated path (`javafx.scene.shape.Path`). A `ScaleTransition` transition targets a node's `scaleX`, `scaleY`, and `scaleZ` properties to resize a node. Translate transition (`TranslateTransition`) targets a node's `translateX`, `translateY`, and `translateZ` properties to move a node across the screen. I've only listed a few common transitions—there are many more transitions I haven't discussed, so I encourage you to look at the Javadocs documentation for subclasses of the `javafx.animation.Transition` class.

Point-and-Click Game Example

To demonstrate the most common JavaFX animation transitions you learned about in this chapter, I created a simple point-and-click game. To give you some background about point-and-click games, in the 90s there were many popular puzzle type games that allowed players to explore and embark on quests by simply using the mouse to point-and-click. One of these famous point-and-click games was called *Myst* (see <http://cyan.com/games/myst>). The game *Myst* begins by throwing the player into a mystical island world to explore and solve various puzzles by examining objects and finding trap doors.

In the point-and-click game shown in Figure 7-8, the player will see animations that demonstrate most of the common JavaFX animation transitions. In the point-and-click example game, the player will be able to click on items to unlock additional animations.

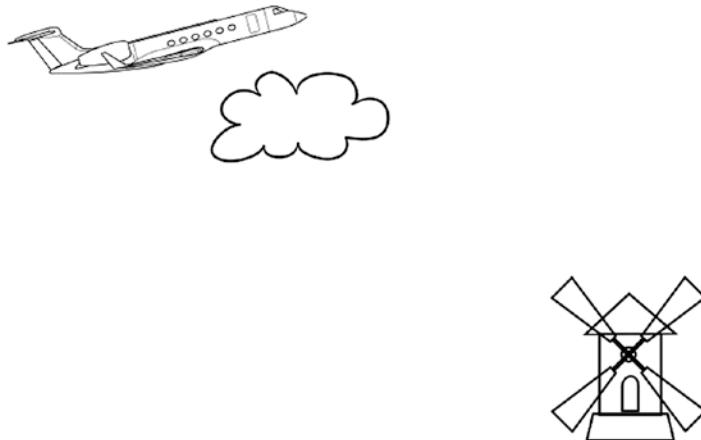


Figure 7-8. A simple point-and-click type game using JavaFX Transition APIs. Depicted in the game are `SVGPath` shapes representing an airplane, a cloud, and a windmill. The rotor blades of the windmill are on a separate `SVGPath` that will demonstrate a `RotationTranslate` animation.

Source Code

The `ClickAndPointGame.java` file is the source code to the point-and-click game shown in Listing 7-28. The game begins by loading SVG paths as JavaFX nodes (`SVGPath`). These SVG nodes are created using drawings downloaded from the web site Wikimedia Commons at <https://commons.wikimedia.org>. There, I found an nice looking airplane from the author named *Kaboldly*. The cloud was from an author by the name of *Thorpe* and the windmill was by an author named *Spedona*. All of these images are under the creative commons license at (<https://creativecommons.org/licenses/by-sa/3.0/deed.en>).

Listing 7-28. Source Code of the ClickAndPointGame.java File Demonstrates Common Animation Transition Classes

```
public class ClickAndPointGame extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("Click And Point Game");
        AnchorPane root = new AnchorPane();
        Scene scene = new Scene(root, 551, 400, Color.WHITE);

        SVGPath plane = createSVGPath("game-assets/plane-svg-path.txt");

        SVGPath windmill = createSVGPath("game-assets/windmill-svg-path.txt");
        AnchorPane.setBottomAnchor(windmill, 50.0);
        AnchorPane.setRightAnchor(windmill, 100.0);

        SVGPath rotorBlades = createSVGPath("game-assets/rotor-blades-svg-path.txt");
        AnchorPane.setBottomAnchor(rotorBlades, 58.0);
        AnchorPane.setRightAnchor(rotorBlades, 86.0);

        // create clouds
        SVGPath cloud1 = createSVGPath("game-assets/cloud-svg-path.txt");

        // Path Transition
        Path flightPath = new Path();
        PathElement startPath = new MoveTo(-200, 100);
        QuadCurveTo quadCurveTo = new QuadCurveTo(100, -50, 500, 100);
        flightPath.getElements()
            .addAll(startPath, quadCurveTo);
        flightPath.setVisible(false);

        PathTransition flyPlane = new PathTransition(Duration.millis(8000),
            flightPath, plane);
        flyPlane.setCycleCount(Animation.INDEFINITE);
        flyPlane.setOrientation(
            PathTransition.OrientationType.ORTHOGONAL_TO_TANGENT);

        // Rotation Transition
        RotateTransition rotateBlade = new RotateTransition(Duration.millis(8000),
            rotorBlades);
        rotateBlade.setCycleCount(Animation.INDEFINITE);
        rotateBlade.setFromAngle(0);
        rotateBlade.setToAngle(360);

        // Scale Transition
        ScaleTransition scaleTransition = new ScaleTransition(Duration.millis(500),
            plane);
        scaleTransition.setCycleCount(4);
        scaleTransition.setAutoReverse(true);
        scaleTransition.setFromX(1);
        scaleTransition.setFromY(1);
    }
}
```

```

scaleTransition.setByX(1.5);
scaleTransition.setByY(1.5);

// Translate Transition
TranslateTransition moveCloud = new TranslateTransition(Duration.seconds(15),
cloud1);
moveCloud.setFromX(-200);
moveCloud.setFromY(100);
moveCloud.setCycleCount(Animation.INDEFINITE);
moveCloud.setAutoReverse(true);
moveCloud.setToX(scene.getWidth() + 200);

// Fade Transition
FadeTransition fadeCloud = new FadeTransition(Duration.millis(1000),
cloud1);
fadeCloud.setCycleCount(4);
fadeCloud.setFromValue(1);
fadeCloud.setToValue(0);
fadeCloud.setOnFinished(actionEvent -> cloud1.setOpacity(1));

// readjust the end points when the width of the screen changes.
scene.widthProperty().addListener( observable -> {
    quadCurveTo.setControlX(scene.getWidth()/2);
    quadCurveTo.setX(scene.getWidth() + 200);
    flyPlane.playFromStart();

    moveCloud.setToX(scene.getWidth() + 200);
    moveCloud.playFromStart();
});

scene.setOnMouseClicked( mouseEvent -> {
    boolean isCloudClicked = cloud1.getBoundsInParent()
        .contains(mouseEvent.getX(),
                  mouseEvent.getY());
    if (isCloudClicked) {
        if (fadeCloud.getStatus() == Animation.Status.STOPPED) {
            fadeCloud.playFromStart();
        }
    }

    boolean isPlaneClicked = plane.getBoundsInParent()
        .contains(mouseEvent.getX(),
                  mouseEvent.getY());
    if (isPlaneClicked) {
        if (scaleTransition.getStatus() == Animation.Status.STOPPED) {
            scaleTransition.playFromStart();
        }
    }
});

```

```

root.getChildren()
    .addAll(flightPath,
            plane,
            cloud1,
            windmill,
            rotorBlades);

primaryStage.setScene(scene);
primaryStage.setOnShowing( windowEvent -> {
    quadCurveTo.setControlX(scene.getWidth()/2);
    quadCurveTo.setX(scene.getWidth() + 200);
    flyPlane.playFromStart();
    rotateBlade.playFromStart();
    moveCloud.playFromStart();
});
primaryStage.show();

}

private SVGPath createSVGPath(String url) {
    SVGPath svgPath = new SVGPath();
    Task<String> svgLoadWorker = createSVGLoadWorker(url);
    svgLoadWorker.setOnSucceeded(stateEvent -> {
        // apply path info
        svgPath.setContent(svgLoadWorker.getValue());
    });
    new Thread(svgLoadWorker).start();
    return svgPath;
}

protected Task<String> createSVGLoadWorker(String pathDataUrl) {
    return new Task<String>() {
        @Override
        protected String call() throws Exception {
            // On the worker thread...
            String pathData = null;

            InputStream in = this.getClass()
                .getClassLoader()
                .getResourceAsStream(pathDataUrl);

            pathData = new Scanner(in, "UTF-8")
                .useDelimiter("\A").next();

            return pathData;
        }
    };
}

```

```

public static void main(String[] args) {
    launch(args);
}
}

```

How It Works

The code in Listing 7-28 begins by loading various SVG shaped paths from file resources in a directory on the classpath. When loading SVG files, I created a private method called `createSVGPath()`. The `createSVGPath()` method will call the `createSVGLoadWorker()` method to read a text file into a Java string, which will later be returned as content for a JavaFX `SVGPath` node. If you are wondering how I was able to obtain the SVG's shaped path, I simply viewed the SVG file in an editor and extracted the path information of the shapes. I highly recommend using the tool called Inkscape (<https://inkscape.org/en/>), which allows you to manipulate SVG files. Any browser will do to view SVG files by selecting the menu option to view source.

After the code creates the individual `SVGPath` nodes that represent the shapes, the code creates various transition animations. Listing 7-28 shows the code's first transition. It's a `PathTransition` animation with the variable `flightPath`. To simulate the airplane flying from the left side of the scene to the right, the path transition will create JavaFX path elements. The path elements consist of a `MoveTo` and a `QuadCurveTo` instance. The path element `QuadCurveTo` will make the airplane appear to climb altitude, showing the nose up and later lowering its nose as it decreases altitude. When using the `PathTransition`, there are different path orientations. This example uses `OrientationType.ORTHOGONAL_TO_TANGENT` to keep the node (`SVGPath`) object perpendicular to the path's tangent point on the curve.

Next, the code creates a `RotateTransition` targeting the windmill's rotor blades to create a rotate animation. The rotation translation is set up to run indefinitely.

Another animation applied to the airplane is a `ScaleTransition` animation. The scale transition occurs when the player clicks on the airplane. When the user clicks on the airplane, it will appear to enlarge and decrease four times, as set by the cycle count property.

To demonstrate the `TranslateTransition` animation, the cloud shape's x translate property will be targeted. The cloud will appear to move from left to right indefinitely via the `setCycleCount(Animation.INDEFINITE)` method.

The last and final transition is the `FadeTransition` animation, which will be applied to the cloud shape's opacity property. This fading in and out animation effect will occur when the player uses the mouse click onto the cloud shape.

In closing, the code creates a mouse click listener to determine if the player has clicked inside an SVG shape. The mouse click code uses the bounds to determine if the mouse pointer coordinates lie inside of the shape to then take the appropriate action. In the case of the airplane, the scale transition effect is applied when the user clicks on the plane. When the cloud is clicked, the fade transition animation is triggered.

Now that you know how to use these simple transitions, you could probably create the next cool point-and-click game! However, you may need more complex animations by chaining together the simple ones you've learned about thus far. Next, you will learn about compound transitions that will allow you to execute many animations in sequence or in parallel.

Compound Transitions

What if you wanted to have multiple animations run in sequence or in parallel? The JavaFX Animation API provides the `SequentialTransition` and `ParallelTransition` classes. A `SequentialTransition`, as its name suggests, allows you to run many animations of type (`javafx.animation.Transition`) in sequence while the `ParallelTransition` class can run many animations in parallel. The following code snippet takes two `FadeTransition` objects to be run in sequence.

```
SequentialTransition seqTransition = new SequentialTransition(fadeOut, fadeIn);
seqTransition.playFromStart();
```

To create a parallel transitions, the following code snippet will perform a translate and rotate on a shape concurrently.

```
ParallelTransition parallelTransition = new ParallelTransition(translateRight, rotateRight);
parallelTransition.playFromStart();
```

This chapter ends with an example of enhancing the Photo Viewer application slightly by using a sequential transition comprised of two `FadeTransition` animations.

PhotoViewer2 Example

To demonstrate a compound transition, I've enhanced the Photo Viewer example mentioned earlier in the chapter to create a fade-out and fade-in effect when the user views the previous or next image. To view the full source code, take a look at the `PhotoViewer2.java` file at the book's website.

The code is quite simple because the `PhotoViewer2` class simply extends from the `PhotoViewer` class and overrides the `loadAndDisplayImage()` method. The overridden method changes the old behavior by adding the fading effect when the image is finished loading, as shown in Listing 7-29. Once the image is successfully loaded, there is a call to a private `transitionByFading()` method to return a `SequentialTransition` animation. After the return of a `SequentialTransition`, the animation is invoked.

Listing 7-29. The Method `loadAndDisplayImage()` Will Create a Worker Task to Load the Next Image to be Displayed

```
protected void loadAndDisplayImage(ProgressIndicator progressIndicator) {
    if (buttonPanel.getCurrentIndex() < 0) return;

    final ImageInfo imageInfo = buttonPanel.getCurrentImageInfo();

    // show spinner while image is loading
    progressIndicator.setVisible(true);

    Task<Image> loadImage = createWorker(imageInfo.getUrl());

    // after loading has succeeded apply image info
    loadImage.setOnSucceeded(workerStateEvent -> {

        try {
            Image nextImage = loadImage.get();
            SequentialTransition fadeIntoNext = transitionByFading(nextImage, imageInfo);
            fadeIntoNext.playFromStart();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        } finally {
            // hide progress indicator
        }
    });
}
```

```

        progressIndicator.setVisible(false);
    }

});

// any failure turn off spinner
loadImage.setOnFailed(workerStateEvent ->
    progressIndicator.setVisible(false));

executorService.submit(loadImage);
}

```

This code creates a worker *task* to load the image on a background thread. Once the image is loaded successfully, a call to the private method `transitionByFading()` will create a sequential transition that will contain two transitions. As shown in Listing 7-30, the method `transitionByFading()` will return a `SequentialTransition` instance to be executed.

Listing 7-30. The `transitionByFading()` is a private Factory Method That Creates a `SequentialTransition` Consisting of Two `FadeTransition` Animations Responsible for Fading Out the Currently Displayed Image and Fading In the Next Image to be Displayed

```

private SequentialTransition transitionByFading(Image nextImage,
                                              ImageInfo imageInfo) {
    // fade out image view node
    FadeTransition fadeOut =
        new FadeTransition(Duration.millis(500), currentViewImage);
    fadeOut.setFromValue(1.0);
    fadeOut.setToValue(0.0);
    fadeOut.setOnFinished(actionEvent -> {
        currentViewImage.setImage(nextImage);
        // Rotate image view
        rotateImageView(imageInfo.getDegrees());

        // Apply color adjust
        colorAdjust = imageInfo.getColorAdjust();
        currentViewImage.setEffect(colorAdjust);

        // update the menu items containing slider controls
        updateSliders();
    });
    // fade in image view node
    FadeTransition fadeIn =
        new FadeTransition(Duration.millis(500), currentViewImage);
    fadeIn.setFromValue(0.0);
    fadeIn.setToValue(1.0);
}

```

```
// fade out image view, swap image and fade in image view
SequentialTransition seqTransition =
    new SequentialTransition(fadeOut, fadeIn);
return seqTransition;
}
```

Listing 7-30 implements the steps into the sequential transition. When the sequential transition is executed, the first transition (`fadeOut`) will fade out the `currentViewImage` (`ImageView`) node containing the current image. The `fadeOut` transition will get handler code to respond when an `OnFinished` event is raised. In other words, when the fade out transition is finished, the code block will set the next image, rotate the image view, adjust the colors, and update the slider controls respectively. After the first transition is run, the second transition gets run to perform the fade-in effect to display the new (next) image.

Assuming you've got this far, you should now have a basic understanding of JavaFX graphics programming.

Summary

In this chapter, you learned how to load images using the `javafx.scene.image.Image` class' constructor, which accepts the standard URL specification format of files located locally, on the classpath, or on a remote web server. Then you got a chance to implement a Photo Viewer application with drag-and-drop capability to load and display images. You also learned about rotation and color adjustments you can make to an image view node.

Next, you looked at the fundamentals of JavaFX's animation API to create key values, keyframes, and timelines. After learning about the fundamental concepts of animating any node using timelines, you looked at convenient animation classes called *transitions*. Using common transitions, you saw a simple point-and-click game example that demonstrated the following transitions: path, rotate, scale, translate, and fade. Finally, you ended the chapter by enhancing the Photo Viewer with a `SequentialTransition` animation to create a compound fade effect as the user views the next or previous image.

CHAPTER 8



JavaFX Printing

Many desktop applications such as word processors and spreadsheets allow users to print documents via a printer or copier. In this chapter, you'll learn how to send HTML documents and JavaFX nodes to a printer. JavaFX printing APIs (`javafx.print`) have been around since JavaFX 8, and they allow application developers to query available printers, set up printers, and generate print jobs. In this chapter, you explore JavaFX's printing APIs and end the chapter with a Web browser-like application to scale and print web content.

In this chapter, you learn about the following:

- Query available printers
- Discover a printer's attributes and features
- Programmatically print (with a printer dialog)
- Show the Printer dialog to configure a print job
- Scale and print a JavaFX node
- Print an HTML web page spanning multiple pages

JavaFX Printing

Before we get started with the JavaFX print APIs, I want to whet your appetite by showing you how easy it is to print something with very few lines of code. JavaFX printing at its simplest form allows you to print any visible JavaFX node, as shown in Listing 8-1.

Listing 8-1. Printing a JavaFX Node Using the `PrinterJob` Class

```
PrinterJob job = PrinterJob.createPrinterJob();
job.showPrintDialog(primaryStage);
boolean success = job.printPage(circle);
if (success) {
    job.endJob();
}
```

The goal in Listing 8-1 is to print a JavaFX `Circle` node to the printer. Listing 8-1 shows the creation of a print job (`PrinterJob`) followed by a call to the `showPrintDialog()` method, which launches a dialog window allowing you to configure the current print job, as shown in Figure 8-1.

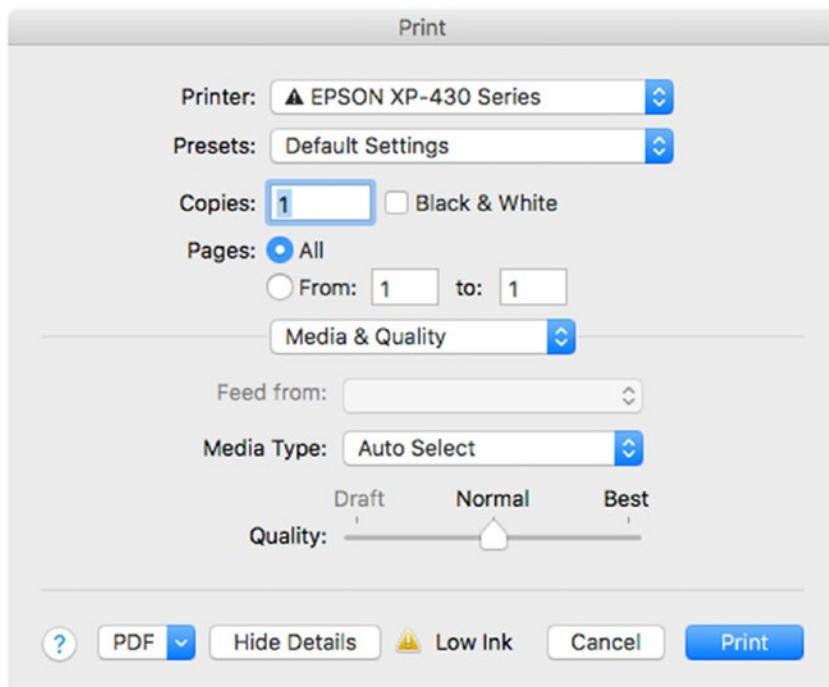


Figure 8-1. A Print dialog window to configure the current print job

After configuring the printer settings, you click on the Print button to begin the printing process. The code will invoke the print job's `printPage()` method to send the job to the printer. After the call to the `printPage()` method a `Boolean` value is returned to signify whether the print job was successful or not. If the printing is successful (`true`), the code finishes the print job by calling the `endJob()` method on the printer job object.

The output from Listing 8-1 should print out a JavaFX circle node onto a printer. Assuming the circle is positioned within the printer's printable width, you should see a single printed page with a circle shape. Later, you learn to use the APIs to query the printer for its printable width and height.

The output of Listing 8-1 looks quite simple to implement; however, there is more than meets the eye. You are probably wondering, “How do you print a node with content that spans multiple pages?”

Since the code simply prints onto a single page, you'll also find out that if a node is larger (outside) than the printable width and height of the printer's layout, those areas won't be printed onto the physical page (Paper). Having said this, you will need to implement a strategy to either resize the node's content or cut up the node's content into multiple nodes to be printed onto each page.

In this chapter, you will primarily learn how to configure a print job and print a single node such as an image or a shape node. However, if you are ambitious enough and willing to create an actual word processor or a greeting card application, you will inevitably want to print content that spans over multiple pages. Because of this, you will likely have to do the math yourself. Basically, this math would involve calculating lines of text or determining the width and height of content regions.

Although creating a word processor capable of printing multiple pages is rather ambitious for a chapter, I will at least provide skeleton code to get you started. As shown in Listing 8-2, the skeleton code is stubbed out so that you can provide helper code (the `MyPrinterHelper.createPages()` method), which will do the math, so to speak, and return a list of nodes that will eventually get queued up for printing.

Listing 8-2. Skeleton Code Stubbed Out to Print a Node That Spans Multiple Pages

```
Printer printer = Printer.getDefaultPrinter();
PageLayout layout = printer.getDefaultPageLayout();

double printWidth = layout.getPrintableWidth();
double printHeight = layout.getPrintableHeight();

PrinterJob job = PrinterJob.createPrinterJob(printer);

// Assuming you have a Node that is larger than your printable area.
TableView myLargeJFXNode = ... // large node spanning multiple pages.

// Create a helper function to return a list of
// nodes representing a page to be printed.
List<Node> pages = MyPrinterHelper.createPages(myLargeJFXNode, printWidth, printHeight);

pages.forEach(job::printPage);
job.endJob();
```

In Listing 8-2, the code begins by obtaining layout information to determine the printable width and height. After obtaining the printable width and height, the code will call the `MyPrinterHelper.createPages()` method to return a list of nodes to print. The `createPages()` method is responsible for calculating how to divide the target node into a list of nodes that will fit the printable print layout. The code statement calling the method `MyPrinterHelper.createPages()` receives the following parameters: `targetNode`, `printWidth`, and `printHeight`.

Once each page is prepared as a node, you can queue them up for printing. Depending on the kind of content to be printed, you will have to decide how you want to break sections into nodes to be queued up for printing. Since it depends on the kind of content to be printed, it is really up to you to decide how you want to design individual nodes. One example is to print a sheet of mailing labels based on a `ListView` node's `getItems()`.

If you feel that creating your own function to break apart content into printable nodes is too difficult, have no fear, JavaFX web engine is here. Later, you will see an example application that will load an HTML web page to be printed as multiple pages. Without jumping ahead, the `javafx.scene.web.WebEngine` node has a convenience method `print()` that will send a web page to the printer. What's nice about the `WebEngine`'s print ability is that it can print without displaying the `WebView` node on the JavaFX Scene graph. This allows you to create a JavaFX application window (`Stage`) that can send HTML documents to the printer in the background. It is important to note that the JavaFX print APIs do not support a true *headless* print service as a background process without having to launch a JavaFX application window. In order to print using JavaFX print APIs, a JavaFX application thread must be running.

Now that you have a taste for how simple JavaFX printing can be, next up you will take a look at JavaFX's print APIs in more detail to programmatically configure printer attributes and print jobs.

JavaFX Print APIs

Let's take a look at the main classes that comprise the JavaFX print APIs. Table 8-1 lists the main print classes and includes a brief description according to the Java 8 Javadoc documentation ([javafx.print](#)).

Table 8-1. The JavaFX Printing API Classes and Their Descriptions

Class/Enum	Description
JobSettings	The JobSettings class encapsulates most of the configuration of a print job.
PageLayout	A PageLayout encapsulates the information needed to lay out content.
PageRange	A PageRange is used to select or constrain the job print stream pages to print.
Paper	A class that encapsulates the size of paper media as used by printers.
PaperSource	A PaperSource is the input tray to be used for the Paper.
Printer	A Printer instance represents the destination for a print job.
PrinterAttributes	This class encapsulates the attributes of a printer that relate to its job printing capabilities and other attributes.
PrinterJob	PrinterJob is the starting place for JavaFX scenegraph printing.
PrintResolution	Class to represent a supported device resolution of a printer in the feed and crossfeed directions in dots-per-inch (DPI).
Collation	The enum class Collation specifies whether or not media sheets of each copy of a printed document in a job are to be in sequence.
PageOrientation	An enum class that specifies the orientation of the media sheet for printed pages.
PrintColor	An enum class that describes whether printing should be monochrome or color.
Printer.MarginType	The MarginType (enum class) is used to determine the printable area of a PageLayout.
PrinterJob.JobStatus	An enum class used in reporting status of a print job.
PrintQuality	An enum class to describe printing quality setting.
PrintSides	An enum class to enumerate the possible duplex (two-sided) printing modes.

Printer and PrinterJob

You might feel overwhelmed by the numerous classes shown in Table 8-1; however, all you have to do is remember the `Printer` and `PrinterJob` classes. Figure 8-2 depicts a high-level diagram showing the relationships between the important JavaFX print API classes.

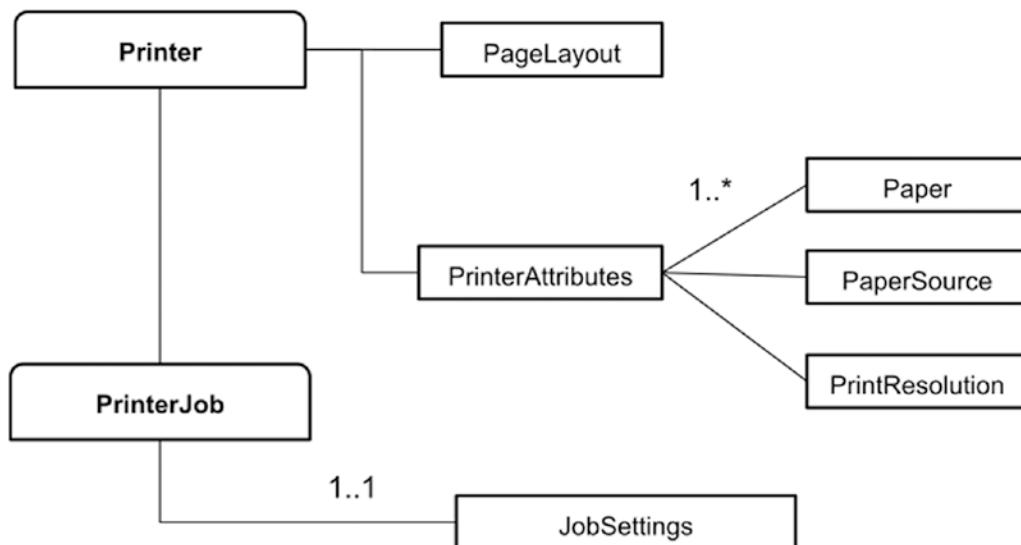


Figure 8-2. A UML diagram of the main JavaFX print classes

For sake of space, the UML diagram shown in Figure 8-2 is showing just a few of the printer attributes of the `PrinterAttributes` object, instead of all of them. You'll notice the diagram displays just a few printer attributes having one-to-many `Paper`, `PaperSource`, and `PrintResolution` objects. However, later you will see example code that outputs all of your printer's supported attribute. For now, let's look at methods to query printers.

To easily get started on querying information, you can focus your attention on the `Printer` and `PrinterJob` classes. Both `Printer` and `PrinterJob` classes contain *static methods* to query printer information and create print jobs, respectively.

To list the available printers, you can invoke the `static method Printer.getAllPrinters()`, as shown in Listing 8-3.

Listing 8-3. Listing the Available Printers from the `Printer` Class

```
Printer.getAllPrinters()
    .forEach(System.out::println);
```

The following is the output from Listing 8-3. It shows a list of printers that are available on my Apple Macbook Pro. Depending on your OS environment and connected printers, you will likely see different results.

```
Printer EPSON XP-430 Series
Printer S300-S400 Series
```

Another convenient static method on the `Printer` class is `getDefaultValue()` to obtain your system's default printer. Depending on your desktop operating system, the user can go to `Printer/Scanner` settings to set up the default printer if one has not been already set. On the Windows operating system, you would navigate to your control panel's `Settings > Devices > <Printer>`. On a MacOS system, navigate to `System Preferences > Printers and Scanners`. On both OSes, you will right-click the selected `<Printer>` to set it as the default printer.

After obtaining a `Printer` object, you can invoke the `getPrinterAttributes()` method that returns a `PrinterAttributes` object to retrieve a printer's supported features. The `PrinterAttributes` object will contain various methods to determine supported orientations, paper types, paper sources, or print resolutions.

Query Printer Attributes

Recall that the UML diagram in Figure 8-2 shows only a few of the attributes from the `PrinterAttributes` object. On the other hand, Listing 8-4 shows the code that lists all of a default printer's attributes. By invoking the `getPrinterAttributes()` method on the printer object (`defaultPrinter`), the default printer's page layout along with all of its supported printer attributes can be queried and dumped to the console.

Listing 8-4. Code to Output a Printer's Default Page Layout and All of Its Supported Attributes.

```
Printer.getAllPrinters().forEach(System.out::println);
Printer defaultPrinter = Printer.getDefaultPrinter();
PrinterAttributes attributes = defaultPrinter.getPrinterAttributes();
System.out.println("-----");
System.out.println("Default Print Layout : ");
System.out.printf(" %s%n", defaultPrinter.getDefaultPageLayout() );
System.out.printf(" printable width: %f%n", defaultPrinter.getDefaultPageLayout().
getPrintableWidth() );
System.out.printf(" printable height: %f%n", defaultPrinter.getDefaultPageLayout().
getPrintableHeight() );
System.out.println("-----");
System.out.println("Supported Orientations : ");
attributes.getSupportedPageOrientations()
    .forEach(System.out::println);

System.out.println("-----");
System.out.println("Supported Collations : ");
attributes.getSupportedCollations()
    .forEach( collation -> System.out.printf(" %s%n", collation));

System.out.println("-----");
System.out.println("Supported Paper types : ");
attributes.getSupportedPapers()
    .forEach( paper -> System.out.printf(" %s%n", paper));

System.out.println("-----");
System.out.println("Supported Paper Sources : ");
attributes.getSupportedPaperSources()
    .forEach( paperSource -> System.out.printf(" %s%n", paperSource));

System.out.println("-----");
System.out.println("Supported Print Colors : ");
attributes.getSupportedPrintColors()
    .forEach( paperColor -> System.out.printf(" %s%n", paperColor));

System.out.println("-----");
System.out.println("Supported Print Quality types : ");
attributes.getSupportedPrintQuality()
    .forEach( printQuality -> System.out.printf(" %s%n", printQuality));
```

```
System.out.println("-----");
System.out.println("Supported Print Resolutions : ");
attributes.getSupportedPrintResolutions()
    .forEach( printRez -> System.out.printf(" %s%n", printRez));

System.out.println("-----");
System.out.println("Supported Print Sides: ");
attributes.getSupportedPrintSides()
    .forEach( printSize -> System.out.printf(" %s%n", printSize));
```

The following is the output from running Listing 8-4.

```
Printer EPSON XP-430 Series
Printer S300-S400 Series
-----
Default Print Layout :
Paper=Paper: Letter size=8.5x11.0 INCH Orient=PORTRAIT leftMargin=54.0 rightMargin=54.0
topMargin=54.0 bottomMargin=54.0
printable width: 504.000000
printable height: 684.000000
-----
Supported Orientations :
PORTRAIT
LANDSCAPE
REVERSE_PORTRAIT
REVERSE_LANDSCAPE
-----
Supported Collations :
-----
Supported Paper types :
Paper: 4x6 size=101.6x152.4 MM
Paper: 8x10 size=8.0x10.0 INCH
Paper: A4 size=210.0x297.0 MM
Paper: A6 size=105.0x148.0 MM
Paper: Legal size=8.4x14.0 INCH
Paper: Letter size=8.5x11.0 INCH
Paper: Number 10 Envelope size=4.125x9.5 INCH
Paper: custom_z_101.6x180.6mm size=101.6x180.6 MM
Paper: invoice size=139.7x215.9 MM
Paper: iso-b7 size=88.0x125.0 MM
Paper: na-5x7 size=127.0x177.8 MM
-----
Supported Paper Sources :
-----
Supported Print Colors :
COLOR
-----
Supported Print Quality types :
NORMAL
-----
```

Supported Print Resolutions :
 Feed res=300dpi. Cross Feed res=300dpi.

Supported Print Sides:
 ONE_SIDED

You've seen the default printer's attributes and supported features, so you can now configure a print job. In the next section, you will learn how to configure and print documents programmatically. The beginning of the chapter explained how to show the printer configuration dialog prior to printing. Next, you learn how to manually configure the printer without a configuration dialog window.

Configuring a Print Job

Now that you know how to query a printer for attributes, you will want to configure the printer slightly differently than the default printer settings. Let's say you have a hypothetical use case where you are trying to conserve on colored ink and paper. In other words, you want to use less black ink, print on both sides, and print using landscape page orientation.

Based on the JavaFX print APIs, a print job will be configured with the following:

- Use one color (`PrintColor.MONOCHROME`)
- Print on both sides of the paper (`PrintSides.DUPLEX`)
- Use a lower print quality (`PrintQuality.LOW`)
- Print using a landscape page orientation (`PageOrientation.LANDSCAPE`)

To achieve this use-case scenario, you will start by calling the method `createPrinterJob()` from the `PrinterJob` class. This method returns a `PrinterJob` object enabling you to obtain and change print job settings via the `getJobSettings()` method. Listing 8-5 shows the code that creates a `PrinterJob` object, then obtains the `JobSettings` object to set up various attributes. To see the full source code for Listing 8-5, refer to the book's source code and Java file named `SaveInkAndTrees.java`.

Listing 8-5. Configuring a Print job's Color, Sides, Quality, and Page Layout

```
PrinterJob job = PrinterJob.createPrinterJob();
JobSettings jobSettings = job.getJobSettings();

jobSettings.setPrintColor(PrintColor.MONOCHROME);
jobSettings.setPrintSides(PrintSides.DUPLEX);
jobSettings.setPrintQuality(PrintQuality.LOW);

PageLayout pageLayout = printer.createPageLayout(Paper.NA_LETTER, PageOrientation.LANDSCAPE,
Printer.MarginType.DEFAULT);

jobSettings.setPageLayout(pageLayout);
System.out.println(">>> jobSettings :\n" + jobSettings);
```

■ Warning A `JobSettings` object may not be modified the way you intended due to the fact of the underlying printer's lack of supported attributes that get mapped to JavaFX related print enum values such as `PrinterColor.MONOCHROME`. You should query the printer's supported attributes before programmatically modifying `JobSettings`.

In Listing 8-5, after obtaining the job settings object, the code begins to call setters for the `color`, `sides`, and `quality` methods. Job settings attributes' can only be modified if they are in the list of support attributes. Refer to the Javadoc documentation for all the available enumerated values of the `PrinterColor`, `PrintSides`, and `PrintQuality` classes. Also, it's important to query supported attributes before setting them because, if you try to set an attribute that isn't supported, the system will take the attribute's default values.

Next, the code creates a page layout via the `createPageLayout()` method. The method receives the following parameters:

- `Paper.NA LETTER`
- `PageOrientation.LANDSCAPE`
- `Printer.MarginType.DEFAULT`

According to the Javadoc documentation, the `Printer.MarginType.DEFAULT` is a default of 0.75 inch margin on all sides. This default is considered to be a common margin by all known printers. However, this may be adjusted if the paper is too small, to ensure that the margins are not more than 50% of the smaller dimension. Applications that do expect to deal with such small media should specify the required margins explicitly. In the unlikely event that the hardware margin is larger than 0.75", it will be adjusted to that same hardware minimum on all sides.

The output of Listing 8-5 is shown here. Since this is my printer at home, I noticed some print attributes did not get modified appropriately such as the sides, print color, and print quality. My printer should have been set with double sided, monochrome, and low quality. You are probably wondering what is going on.

```
>>> JobSettings:
Collation = UNCOLLATED
Copies = 1
Sides = ONE_SIDED
JobName = JavaFX Print Job
Page ranges = null
Print color = COLOR
Print quality = NORMAL
Print resolution = Feed res=300dpi. Cross Feed res=300dpi.
Paper source = Paper source : Automatic
Page layout = Paper=Paper: Letter size=8.5x11.0 INCH Orient=LANDSCAPE leftMargin=54.0
rightMargin=54.0 topMargin=54.0 bottomMargin=54.0
```

Not to be alarmed, but in Listing 8-5 while making job settings changes, I noticed some of the settings didn't get modified. Basically this is due to the underlying printer's lack of supported attributes that get mapped to JavaFX printer enums and print attribute classes. By printing the job settings object, you can compare your attributes against the supported attributes. I provided code in Listing 8-6 to check for supported attributes based on the earlier print scenario. If the print attributes you are looking for are not supported, the code in Listing 8-6 will launch the printer dialog. This will allow the native print dialog window to let the user modify attributes before printing.

Listing 8-6. Determining if a Printer's Print Attributes Are Supported Before Sending Job for Printing

```

Printer printer = job.getPrinter();
PrinterAttributes attr = printer.getPrinterAttributes();
boolean supported = attr.getSupportedPrintColors()
    .contains(PrintColor.MONOCHROME) &&
    attr.getSupportedPrintSides()
        .contains(PrintSides.DUPLEX) &&
    attr.getSupportedPrintQuality()
        .contains(PrintQuality.LOW) &&
    attr.getSupportedPapers()
        .contains(Paper.NA_LETTER) &&
    attr.getSupportedPageOrientations()
        .contains(PageOrientation.LANDSCAPE);
if (!supported) {
    job.showPrintDialog(primaryStage);
}
// Rest of the print code.

```

Now that you know how to configure job settings and create documents to be sent to the printer, let's look at how to print HTML5-based content.

Printing a Web Page

As mentioned earlier, it's not exactly easy to print content spanning multiple pages. However, the JavaFX APIs do provide a convenient method called `print()` on the `WebEngine` object. It allows you to print HTML5 content. The `WebEngine` object is a non-visual JavaFX object capable of loading HTML documents from remote web servers. Listing 8-7 shows a minimum amount of code that will load and print a web page to the default printer. If the document spans multiple pages, the printer will continue to print all of the remaining pages.

Listing 8-7. Printing Web Content Using JavaFX's `WebEngine` Object

```

WebEngine webEngine = new WebEngine();

webEngine.getLoadWorker()
    .stateProperty()
    .addListener( (ov, oldState, newState) -> {
        if (newState == State.SUCCEEDED) {
            PrinterJob job = PrinterJob.createPrinterJob();
            if (job != null) {
                webEngine.print(job);
                job.endJob();
            }
        }
    });
webEngine.load("https://en.wikipedia.org/wiki/Tachyon");

```

This code will provide a callback to listen to state changes of the load process of the web engine. Upon success (`State.SUCCEEDED`), a print job is created and is passed into the `WebEngine` object's `print()` method.

Hopefully, you can see how amazingly simple it is to send HTML content to the printer. Now let's look at an example application.

Example WebDocPrinter Application

Have you ever needed to quickly print an airline ticket or an invoice after an online purchase? Well, here is an example application that I've created resembling a browser by allowing users to load and display a web site based on a valid URL. This browser-like application is called WebDocPrinter, as shown in Figure 8-3. It supports two print modes: Node Only and Whole Web Document. By default, the Node Only is set to allow the user to print the WebView node only. Node Only will only print content within the red bounding box. The Whole Web Document print mode will print the entire HTML document, even if it spans multiple pages. The Whole Web Document print mode scales the content to fit the printable area.



Figure 8-3. The WebDocPrinter application supports two print modes: Node Only and Whole Web Document

The application has the following UI elements:

- Menus: Two print modes called Node Only and Whole Web Document
- TextField: An address bar to enter a valid URL
- Button: A print button
- Slider: A slider control that adjusts the zoom level when displaying the WebView node

- **Label:** Display label to display the zoom level as a percentage
- **WebView:** The displaying of the HTML document
- **Path:** A rectangle path with a red stroke outlining the printable region

The WebDocPrinter application supports two print modes: Node Only and Whole Web Document. The WebDocPrinter application demonstrates the printing of a single node or the printing of an entire web page having multiple pages. Using the Whole Web Document print automatically scales the content to fit the printable area.

After entering the URL on the address line (TextField), the user will press the Enter key, thereby triggering the application to fetch the web page to be displayed onto the WebView node. By default, the menu bar's print mode, as shown in Figure 8-4, is set to Node Only print mode, which means the application will only print the WebView node on one page.

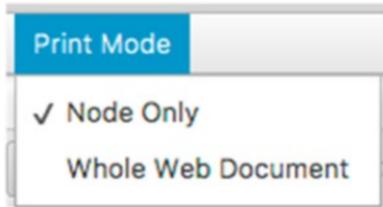


Figure 8-4. The two print modes are Node Only and Whole Web Document

When printing a JavaFX WebView node, the printer will only print to one page by default. By using the Node Only mode, a red box outline will appear over the display area to denote the printer's printable layout region onto the physical paper, as shown in Figure 8-3. Using the Node Only print mode, anything outside the red bounding box will not be printed, as discussed earlier.

The second print mode is Whole Web Document, which will print the entire HTML web page using the convenient method `print()` from the `WebEngine` object. Again, if the document spans multiple pages, they'll be sent to the printer.

Finally, the last feature of the WebDocPrinter application is a slider control that allows users to scale or zoom in/out to resize the web page's content, as shown in Figure 8-5.

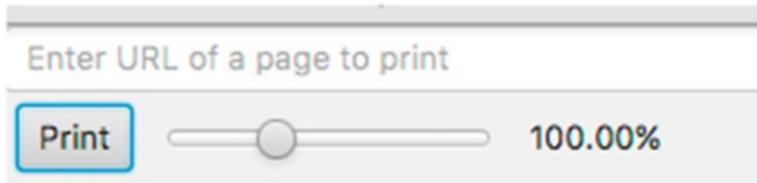


Figure 8-5. The URL address line allows the user to fetch a web page. The print button sends the displayed page to the printer. The slider control zooms in or out to scale the web page to fit into the print width.

The zoom feature used during the Node Only print mode will assist the user to make the content fit within the printable region (red bounding box) before being sent to the printer, as shown in Figure 8-3. The zoom's minimum value is 5% and the maximum is 300%. Again, I want to point out that any content displayed outside of the red bounding box region will not get printed when using the Node Only print mode.

Source Code

Now that you know what WebDocPrinter application is capable of, let's see the source code. Because the actual print code is quite small, I kept all the code in one file, called `WebDocPrinter.java`, as shown in Listing 8-8.

Listing 8-8. The WebDocPrinter Application Source Code

```
/**
 * Allows the user to enter a URL to display an HTML page to be
 * sent to a default printer. Also the application allows the
 * display node to be resize to fit onto the printed page.
 */
public class WebDocPrinter extends Application{
    private static String PRINT_MODE_MENU = "Print Mode";
    private static String NODE_ONLY = "Node Only";
    private static String WHOLE_WEB_DOC = "Whole Web Document";
    public static void main(String[] args) {
        //System.setProperty("jsse.enableSNIExtension", "false");
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) throws Exception {
        // Create the root pane and scene
        primaryStage.setTitle("WebDocPrinter");
        BorderPane root = new BorderPane();
        Scene scene = new Scene(root, 551, 400, Color.WHITE);
        primaryStage.setScene(scene);

        // Create a menu
        MenuBar menuBar = new MenuBar();
        Menu printModeMenu = new Menu("Print Mode");
        ToggleGroup printModeGroup = new ToggleGroup();

        // Node only print mode
        RadioMenuItem printOnePage = new RadioMenuItem(NODE_ONLY);
        printOnePage.setUserData(NODE_ONLY);
        printOnePage.setToggleGroup(printModeGroup);
        printModeGroup.selectToggle(printOnePage);
        printModeMenu.setText(PRINT_MODE_MENU + " (" + NODE_ONLY + ")");

        // Whole web document print mode
        RadioMenuItem multiPages = new RadioMenuItem("Whole Web Document");
        multiPages.setUserData(WHOLE_WEB_DOC);
        multiPages.setToggleGroup(printModeGroup);

        printModeMenu.getItems().addAll(printOnePage, multiPages);
        menuBar.getMenus().add(printModeMenu);
        root.setTop(menuBar);

        BorderPane contentPane = new BorderPane();
        root.setCenter(contentPane);
    }
}
```

```

// Create display area
WebView browserDisplay = new WebView();

// Create a slider to control zoom
Slider zoomSlider = new Slider(.05, 3.0,1.0);
zoomSlider.setBlockIncrement(0.05);
zoomSlider.valueProperty().addListener( listener -> {
    System.out.println("zoom " + browserDisplay.getZoom());
    browserDisplay.setZoom(zoomSlider.getValue());
});

// Label representing the zoom size percentage
Label zoomValueLabel = new Label();
StringConverter sc = new StringConverter<Double>(){
    @Override public Double fromString(String value) {
        // If the value is null or empty string return null
        if (value == null) {
            return null;
        }

        value = value.trim();
        value.replace("%", "");
        if (value.length() < 1) {
            return null;
        }

        return Double.valueOf(value)/100;
    }

    @Override public String toString(Double value) {
        //If the value is null, return empty string
        if (value == null) {
            return "";
        }
        double percent = value.doubleValue() * 100;
        return String.format("%.2f", percent) + "%";
    }
};

// Bind Label's text and the Slider's value property.
Bindings.bindBidirectional(zoomValueLabel.textProperty(),
    zoomSlider.valueProperty(), sc);

// debug information
browserDisplay.widthProperty().addListener( listener -> {
    Printer printer = Printer.getDefaultPrinter();
    System.out.println("printer width: " +
        printer.getDefaultPageLayout().getPrintableWidth());
    System.out.println("width: " +
        browserDisplay.widthProperty().get() );
});

```

```

WebEngine webEngine = browserDisplay.getEngine();
webEngine.getLoadWorker()
    .stateProperty()
    .addListener( (obsValue, oldState, newState) -> {
        if (newState == Worker.State.SUCCEEDED) {
            System.out.println("finished loading webpage: " +
                webEngine.getLocation());
        }
    });
});

// Create an address bar
TextField urlAddressField = new TextField();
urlAddressField.setPromptText("Enter URL of a page to print");
urlAddressField.setOnAction( actionEvent ->
    webEngine.load(urlAddressField.getText()));

// Create the print button
Button printButton = new Button("Print");
printButton.setOnAction(actionEvent -> {

    PrinterJob job = PrinterJob.createPrinterJob();
    if (job != null) {
        System.out.println("starting print job");
        Toggle selected = printModeGroup.getSelectedToggle();
        if (selected != null) {

            String mode = (String) selected.getUserData();
            if (NODE_ONLY.equals(mode)) {
                boolean success = job.printPage(browserDisplay);
                if (success) {
                    job.endJob();
                }
            } else {
                // WHOLE_WEB_DOC
                boolean printIt = job.showPrintDialog(primaryStage);
                if (printIt) {
                    webEngine.print(job);
                    job.endJob();
                }
            }
        }
    }
});

// Assemble print button, zoom slider, zoom label
VBox vBox = new VBox();
HBox hBox = new HBox(10);
hBox.setPadding(new Insets(5));

```

```
hBox.setAlignment(Pos.CENTER_LEFT);
hBox.getChildren().addAll(printButton, zoomSlider, zoomValueLabel);
vBox.getChildren().addAll(urlAddressField, hBox);

contentPane.setTop(vBox);

// Center WebView area
StackPane centerArea = new StackPane(browserDisplay);

// Create the red box denoting print area.
Path printPerimeter = new Path();
Printer printer = Printer.getDefaultPrinter();
double printWidth = printer.getDefaultPageLayout().getPrintableWidth();
double printHeight = printer.getDefaultPageLayout().getPrintableHeight();
PathElement[] corners = {
    new MoveTo(0,0),
    new LineTo(printWidth, 0),
    new LineTo(printWidth, printHeight),
    new LineTo(0, printHeight),
    new ClosePath()
};
printPerimeter.getElements().addAll(corners);
printPerimeter.setStroke(Color.RED);
StackPane.setAlignment(printPerimeter, Pos.TOP_LEFT);
centerArea.getChildren().add(printPerimeter);
contentPane.setCenter(centerArea);

printModeGroup.selectedToggleProperty()
    .addListener((observableValue) -> {

    Toggle selected = printModeGroup.getSelectedToggle();
    if (selected != null) {
        String mode = String.valueOf(selected.getUserData());
        printModeMenu.setText(PRINT_MODE_MENU + " (" + mode + ")");
        if (NODE_ONLY.equals(mode)) {
            printPerimeter.setVisible(true);
        } else {
            printPerimeter.setVisible(false);
        }
    }
});

primaryStage.setOnShown( eventHandler -> {
    printButton.requestFocus();
});

primaryStage.show();
}
}
```

How Does It Work?

In Listing 8-8, note that the `start()` method begins with creating a root pane and scene typical of all JavaFX applications. Next, the code creates the two menu options to allow users to select a print mode. Later in the code, the menu items will add handler code to items when they are selected. After the menus are created, the code sets up the `WebView` node and the zoom *slider control*.

The `WebView` node is the main area to display an *HTML* document. The zoom slider control will add handler code (`InvalidationListener`) to update the `WebView` node's zoom property.

When displaying the zoom value as a percentage, a JavaFX `Label` is updated whenever the slider value changes. Here, you will see how a string is converted to a double using a `StringConverter` class. The `StringConverter` class converts a *double value* from the slider control to a *text value* of a label control having a % percent symbol appended. Once the `StringConverter` object is created and assigned to the variable `sc`, the code will bind properties based on the label's text property (`String`), zoom slider's value (`Double`), and variable `sc`.

The following code snippet binds the properties and variables mentioned.

```
Bindings.bindBidirectional(zoomValueLabel.textProperty(),
    zoomSlider.valueProperty(), sc);
```

The code continues to create the rest of the application by adding a JavaFX `TextField` for a URL address line similar to web browsers. The address line allows the user to enter a valid URL to request a specified web page. The text field has an `onAction` property to set handler code that will then call the web engine's `load()` method. In other words, after the user enters the web address, she will then press the Enter key to trigger the loading of the web page.

Next is the creation of the Print button (`printButton` variable) to print based on the selected print mode. The Print button will determine the print mode before sending data to the printer. When the user chooses Node Only, the code will pass the `WebView` node into the `printPage()` method of the `PrintJob` object. If the user chooses to print the whole document, the code obtains the web engine (`WebEngine`) and passes the print job to the `print()` method. The following snippet of code is from Listing 8-8 and it shows the two print modes again—Node Only and Whole Web document.

```
// Node Only - print mode
job.printPage(browserDisplay);

// Whole Web Doc - print mode
webEngine.print(job);
```

Finishing up with Listing 8-8, the code creates a path outlined as a rectangle consisting of width and height values from the default printer's layout. Lastly, the `printModeGroup` (`ToggleGroup`) variable's `selectedToggleProperty` attaches handler code that toggles the red rectangle shape's `visible` property. Also, when the Print button is clicked, the print mode is determined by calling the `printModeGroup` objects' `getUserObject()` method to obtaining a string. This string represents one of the two string constants: `NODE_ONLY` or `WHOLE_WEB_DOC`.

Summary

In this chapter you began with a simple code snippet that launches a print dialog to configure your printer before printing a JavaFX node. Next, you got acquainted with the JavaFX print APIs by querying and displaying all of a printer's supported features and attributes. After learning about print attributes, you were able to programmatically configure a print job using the `JobSettings` object obtained from a `PrinterJob` instance. While configuring a print job programmatically, you were warned about attributes not getting modified unless the printer supports a particular attribute such as printing double sided or black and white.

After learning about how to query and configure printer attributes, you were able to learn about the convenient `print()` method from the `WebEngine` object. Also, you learned that the `print()` method can print web content that spans multiple pages, and you can automatically scale web content to fit each page. Lastly, you got a chance to look at an example application called `WebDocPrinter.java` that is capable of printing web content in two print modes: Node Only and Whole Web Document. The Node Only print mode prints a portion of the web document by sending the `WebView` node to the printer, while the Whole Web Document print mode prints the whole web page even if it spans multiple pages.

CHAPTER 9



Media and JavaFX

JavaFX provides a media-rich API capable of playing audio and video. The Media API allows developers to incorporate audio and video into their Rich Internet Applications (RIAs). One of the main benefits of the Media API is that it can distribute cross-platform media content via the Web. With a range of devices (tablet, music player, TV, and so on) that need to play multimedia content, the need for a cross-platform API is essential.

Imagine a not-so-distant future where your TV or wall is capable of interacting with you in ways that you've never dreamed possible. For instance, while viewing a movie, you could select items of clothing used in the movie to be immediately purchased, all from the comfort of your home. With this future in mind, developers seek to enhance the interactive qualities of their media-based applications.

In this chapter, you will learn how to play audio and video in an interactive way. Find your seats as audio and video take center stage.

You will learn about the following JavaFX media APIs:

- `javafx.scene.media.Media`
- `javafx.scene.media.MediaPlayer`
- `javafx.scene.media.MediaStatus`
- `javafx.scene.media.MediaView`

Media Events

An event-driven architecture (EDA) is a prominent architectural pattern used to model loosely coupled components and services that pass messages asynchronously. The JavaFX team designed the Media API to be event-driven. In this section, you learn how to interact with media events.

With event-based programming in mind, you will discover nonblocking or callback behaviors when invoking media functions. Instead of typing code directly to a button via an `EventHandler`, you will be implementing code that will respond to the triggering of the media player's `OnXXXX` events, where `XXXX` is the event name.

When responding to media events, you will implement `java.lang.Runnable` functional interfaces (lambda expressions). These functional interfaces are callbacks that are lazily evaluated or invoked at a later time upon a triggered event. For instance, when playing media content, you would create a lambda expression to be set on the `OnReady` event. For an example of playing media based on an event-driven approach, see the following code block:

```
Media media = new Media(url);
MediaPlayer mediaPlayer = new MediaPlayer(media);
Runnable playMusic = () -> mediaPlayer.play();
mediaPlayer.setOnReady(playMusic);
```

As you can see, the `playMusic` variable is assigned to a lambda expression (`Runnable`) to be passed into the media player's `setOnReady()` method. When the `OnReady` event has been encountered, the `playMusic` `Runnable` closure code would be invoked. Although I have not yet discussed how to set up the `Media` and `MediaPlayer` class instances, I wanted to first get you acquainted with these core concepts before moving further, because these concepts are used throughout this chapter.

Table 9-1 shows all the possible media events that are raised to allow the developer to attach `Runnables` (or `EventHandlers`). All the following are `Runnable` functional interfaces except for the `OnMarker` event.

Table 9-1. Media and MediaPlayer Events

Class	On Event Method	On Event Property Method	Description
Media	<code>setOnError()</code>	<code>onErrorProperty()</code>	When an error occurs
MediaPlayer	<code>setOnEndOfMedia()</code>	<code>onEndOfMediaProperty()</code>	When the end of the media play is reached
MediaPlayer	<code>setOnError()</code>	<code>onErrorProperty()</code>	When an error occurs; status becomes <code>MediaPlayer.Status HALTED</code>
MediaPlayer	<code>setOnHalted()</code>	<code>onHaltedProperty()</code>	When the media status changes to <code>HALTED</code>
MediaPlayer	<code>setOnMarker()</code>	<code>onMarkerProperty()</code>	When a <code>Marker</code> event is triggered
MediaPlayer	<code>setOnPaused()</code>	<code>onPausedProperty()</code>	When the status is <code>MediaPlayer.Status PAUSED</code>
MediaPlayer	<code>setOnPlaying()</code>	<code>onPlayingProperty()</code>	When the media is currently playing. When the status is <code>MediaPlayer.Status PLAYING</code>
MediaPlayer	<code>setOnReady()</code>	<code>onReadyProperty()</code>	When the media player is in Ready state. When the status is <code>MediaPlayer.Status READY</code>
MediaPlayer	<code>setOnRepeat()</code>	<code>onRepeatProperty()</code>	Event handler invoked when the player <code>currentTime</code> reaches <code>stopTime</code> and will be repeating. This callback is made prior to seeking back to <code>startTime</code>
MediaPlayer	<code>setOnStalled()</code>	<code>onStalledProperty()</code>	When the media status changes to <code>STALLED</code>
MediaPlayer	<code>setOnStopped()</code>	<code>onStoppedProperty()</code>	When the status stopped is raised. <code>MediaPlayer.Status STOPPED</code>

Playing Audio

JavaFX's media API supports loading audio files with extensions such as .mp3, .wav, and .aiff. Also, new in JavaFX 8 is the ability to play audio in HTTP live streaming format, also known as HLS (file extension .m3u8). HLS is beyond the scope of this book and will not be covered in this chapter, but with some further research, JavaFX could be used to build a live radio broadcast application.

Playing audio files is extremely straightforward in JavaFX. Given a valid URL location to a file, you would instantiate a `javafx.scene.media.Media` class to load a resource. The `Media` object is then passed to a new instance of a `javafx.scene.media.MediaPlayer` object's constructor to create media player controls. The last step is to invoke the media player object's `play()` method when the `OnReady` event is triggered, at this point the media file will begin playing. To automatically play the media file after loading the media player's auto play property can be set to true using the `setAutoPlay()` method. The following code loads and plays an MP3 audio file located on a web server:

```
Media media = new Media("http://some_host/eye_on_it.mp3");
MediaPlayer mediaPlayer = new MediaPlayer(media);
mediaPlayer.setAutoPlay(true);
```

Note When loading media files, make sure the file location is a formatted string that follows the standard URL specification: <https://www.w3.org/Addressing/URL/url-spec.txt> and <https://url.spec.whatwg.org>.

The media file can reside on a web server, in a JAR file, or on a local filesystem as long as the filename string is formatted to follow the standard URL specification.

Note For low-latency playback of audio files, use the `javafx.scene.media.AudioClip` class. A typical scenario is to play a given sound multiple times consecutively, as with sound effects used in games.

An MP3 Player Example

Now that you know how to load and play audio media, let's look at a fun example that involves playing music and displaying a colorful visualization. In this section, you learn how to create an MP3 audio player. Before exploring the example's code in Listing 9-1, take a look at Figure 9-1 for a preview of the audio player's UI. When an audio file is being played, a spectrum display is shown as an area chart. The audio player's controls allows you to see the progress of the media being played and to stop, pause, and play. In Figure 9-1, the background is white; however, the actual code will show things as a black background instead. On printed paper, it is ideal to avoid large, dark regions.

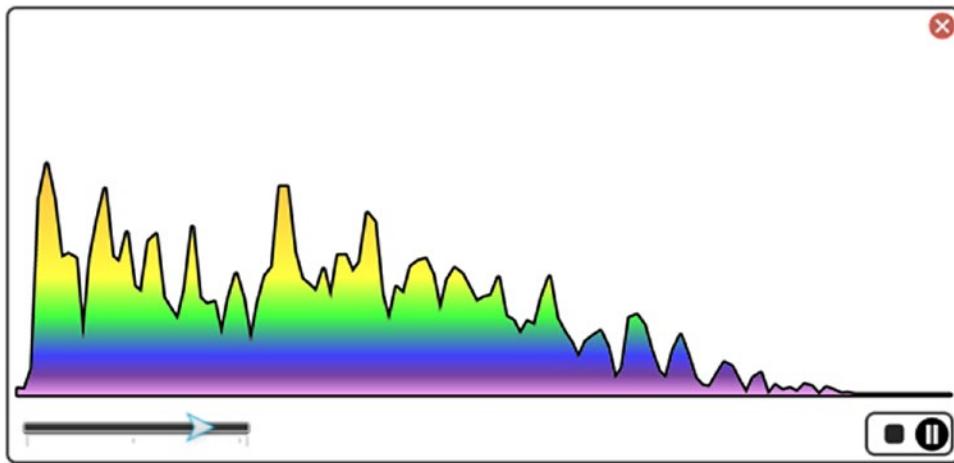


Figure 9-1. A JavaFX audio player

In this example, you will again use the file drag-and-drop metaphor, as you did in Chapter 7 with images. In the same manner, the user navigates to the local filesystem or a browser URL address bar to locate a media file to be dragged and dropped onto the application surface. In the case of the MP3 player, you will locate an audio file with a file format and extension of .mp3, .wav, or .aif to be used in this example.

As you can see in Figure 9-1, the MP3 player displays an area line chart based on the media's audio spectrum information. The audio player has a custom button panel control located in the bottom-right corner. The button panel control lets the user pause, resume, and stop playing music. You'll also notice the seek position slider (aka, the progress and seek position slider) on the bottom-left side, which allows users to watch the progress of the playing of the music, but also seek backward or forward in the media (audio or video). When allowing users to seek backward or forward, the audio or video will need to be paused first.

The Stop, Play, and Pause Buttons

The left image in Figure 9-2 shows the Stop (square) and Play (triangle wedge) buttons. When the Stop button is clicked, the media start time is repositioned to the beginning of the file. When the Pause button is clicked, the media will maintain its current position within the media playback. Also, as the media is paused, the Play button will appear, so that the user can resume the music. The right image in Figure 9-2 shows the button panel containing a Stop and a Pause button (circle with two vertical lines). While the music is currently playing, the user can click the Pause button to pause the music.



Figure 9-2. Custom button panel control with Stop (rectangle), Play (triangle), and Pause (circle) buttons

The Progress and Seek Position Slider Control

Next, you'll notice the progress and seek position slider control in the lower-left corner of the application window, as shown in Figure 9-3. Assuming the media is paused, the seek position slider control allows you to move backward or forward into the media's current time. Also, the slider's "thumb" (arrow) will move from left to right as the media play is progressing.

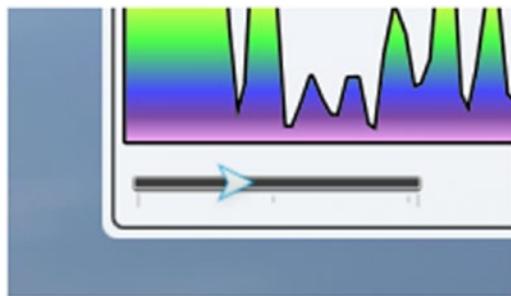


Figure 9-3. The Seek Position slider control allows users to monitor the progress as an audio file is being played

The Close Button

Finally, notice the Close button, shown in Figure 9-4, which simply allows the user to quit the application.

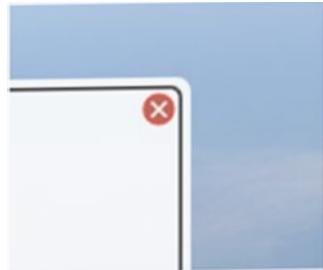


Figure 9-4. The Close button, located in the upper-right corner, allows the user to stop the media player and quit the application

MP3 Audio Player Source Code

Listing 9-1 shows the source code for a JavaFX-based MP3 audio player. The source code from a previous editions of this book was further refactored to break things into methods instead of having one large `start()` method. In addition to the refactoring, I pulled out the code that used a programmatic approach to style JavaFX nodes. Instead of styling JavaFX nodes with `setter()` methods, I simply created a CSS style sheet (declarative style). The style sheet shown in Listing 9-2 contains the CSS selector definitions.

Listing 9-1. An MP3 Audio Player Application (PlayingAudio.java)

```
package com.jfxbe;

import java.net.MalformedURLException;
import javafx.application.*;
import javafx.beans.property.*;
import javafx.beans.value.ChangeListener;
import javafx.event.EventHandler;
import javafx.geometry.Point2D;
import javafx.scene.*;
import javafx.scene.control.Alert;
import javafx.scene.control.ButtonType;
import javafx.scene.control.Slider;
import javafx.scene.input.*;
import javafx.scene.layout.*;
import javafx.scene.media.*;
import javafx.scene.media.MediaPlayer.Status;
import javafx.scene.paint.Color;
import javafx.scene.shape.*;
import javafx.stage.*;
import javafx.util.Duration;

/**
 * Playing Audio using the JavaFX MediaPlayer API.
 *
 * @author carldea
 */
public class PlayingAudio extends Application {

    private MediaPlayer mediaPlayer;
    private Point2D anchorPt;
    private Point2D previousLocation;
    private ChangeListener<Duration> progressListener;
    private BooleanProperty playAndPauseToggle = new SimpleBooleanProperty(true);
    private EventHandler<MouseEvent> mouseEventConsumer = event -> event.consume();

    /**
     * @param args he command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        // Remove native window borders and title bar
        primaryStage.initStyle(StageStyle.TRANSPARENT);

        // Create the application surface or background
        Pane root = new AnchorPane();
        root.setId("app-surface");
```

```
Scene scene = new Scene(root, 551, 270, Color.rgb(0, 0, 0, 0));

// load JavaFX CSS style
scene.getStylesheets()
    .add(getClass().getResource("/playing-audio.css")
        .toExternalForm());
primaryStage.setScene(scene);

// Initialize stage to be movable via mouse
initMovablePlayer(primaryStage);

// Create a Path instance for the area chart
Path chartArea = new Path();
chartArea.setId("chart-area");

// Create the button panel (stop, play and pause)
Node buttonPanel = createButtonPanel(root);
AnchorPane.setRightAnchor(buttonPanel, 3.0);
AnchorPane.setBottomAnchor(buttonPanel, 3.0);

// Create a slider for our progress and seek control
Slider progressSlider = createSlider();
AnchorPane.setLeftAnchor(progressSlider, 2.0);
AnchorPane.setBottomAnchor(progressSlider, 2.0);

// Updates slider as audio/video is progressing (play)
progressListener = (observable, oldValue, newValue) ->
    progressSlider.setValue(newValue.toSeconds());

// Initializing Scene to accept files
// using drag and dropping over the surface to load media
initFileDragNDrop(root);

// Create the close button
Node closeButton = createCloseButton();
AnchorPane.setRightAnchor(closeButton, 2.0);
AnchorPane.setTopAnchor(closeButton, 2.0);

root.getChildren()
    .addAll(chartArea,
            buttonPanel,
            progressSlider,
            closeButton);

primaryStage.centerOnScreen();
primaryStage.show();
}
```

```

/**
 * Initialize the stage to allow the mouse cursor to move the application
 * using dragging.
 * @param primaryStage - The applications primary Stage window.
 */
private void initMovablePlayer(Stage primaryStage) {

    Scene scene = primaryStage.getScene();
    Pane root = (Pane) scene.getRoot();
    root.setPickOnBounds(true);
    // starting initial anchor point
    root.setOnMousePressed(mouseEvent ->
        anchorPt = new Point2D(mouseEvent.getScreenX(),
                               mouseEvent.getScreenY())
    );

    // Dragging the stage by moving its x,y
    // based on the previous location.
    root.setOnMouseDragged(mouseEvent -> {
        if (anchorPt != null && previousLocation != null) {
            primaryStage.setX(previousLocation.getX()
                + mouseEvent.getScreenX()
                - anchorPt.getX());
            primaryStage.setY(previousLocation.getY()
                + mouseEvent.getScreenY()
                - anchorPt.getY());
        }
    });

    // Set the new previous to the current mouse x,y coordinate
    root.setOnMouseReleased(mouseEvent ->
        previousLocation = new Point2D(primaryStage.getX(),
                                       primaryStage.getY())
    );

    // Initialize previousLocation after Stage is shown
    primaryStage.addHandler(WindowEvent.WINDOW_SHOWN,
        (WindowEvent t) -> {
            previousLocation = new Point2D(primaryStage.getX(),
                                           primaryStage.getY());
        });
}

/**
 * Initialize the Drag and Drop ability for media files.
 * @param root - The Scene graph's root pane.
 */
private void initFileDragNDrop(Pane root) {
    // Drag over surface
    root.setOnDragOver(dragEvent -> {
        Dragboard db = dragEvent.getDragboard();

```

```

        if (db.hasFiles() || db.hasUrl()) {
            dragEvent.acceptTransferModes(TransferMode.LINK);
        } else {
            dragEvent.consume();
        }
    });

    // Dropping over surface
    root.setOnDragDropped(dragEvent -> {
        Dragboard db = dragEvent.getDragboard();
        boolean success = false;
        String filePath = null;
        if (db.hasFiles()) {
            success = true;
            if (db.getFiles().size() > 0) {
                try {
                    // obtain file and play media
                    filePath = db.getFiles()
                        .get(0)
                        .toURI().toURL().toString();
                    playMedia(filePath, root);
                } catch (MalformedURLException ex) {
                    ex.printStackTrace();
                }
            }
        } else {
            // audio file from some host or jar
            playMedia(db.getUrl(), root);
            success = true;
        }
        dragEvent.setDropCompleted(success);
        dragEvent.consume();
    });
}

/**
 * Creates a node containing the audio player's
 * stop, pause and play buttons.
 *
 * @param root The Scene graphs root pane.
 * @return Node A button panel having play,
 *         pause and stop buttons.
 */
private Node createButtonPanel(Pane root) {

    // create button control panel
    FlowPane buttonPanel = new FlowPane();
    buttonPanel.setId("button-panel");
}

```

```
// stop button control
Node stopButton = new Rectangle(10, 10);
stopButton.setId("stop-button");
stopButton.setOnMouseClicked(mouseEvent -> {
    if (mediaPlayer != null) {
        updatePlayAndPauseButtons(true, root);
        if (mediaPlayer.getStatus() == Status.PLAYING ||
            mediaPlayer.getStatus() == Status.PAUSED) {

            mediaPlayer.stop();
        }
        playAndPauseToggle.set(false);
    }
});

// Toggle Button containing a Play and Pause button
StackPane playPauseToggleButton = new StackPane();

// Play button control
Arc playButton = new Arc(12, // center x
    16, // center y
    15, // radius x
    15, // radius y
    150, // start angle
    60); // length
playButton.setId("play-button");
playButton.setType(ArcType.ROUND);
playButton.setMouseTransparent(true);

// Pause control
Group pauseButton = new Group();
pauseButton.setId("pause-button");
Circle pauseBackground = new Circle(12, 16, 10);
pauseBackground.getStyleClass()
    .addAll("pause-circle");

Line firstLine = new Line(6, // start x
    6, // start y
    6, // end x
    14); // end y
firstLine.getStyleClass()
    .addAll("pause-line", "first-line");

Line secondLine = new Line(6, // start x
    6, // start y
    6, // end x
    14); // end y
secondLine.getStyleClass()
    .addAll("pause-line", "second-line");
```

```
pauseButton.getChildren()
    .addAll(pauseBackground, firstLine, secondLine);
pauseButton.setMouseTransparent(true);

playPauseToggleButton.getChildren()
    .addAll(playButton, pauseButton);
playPauseToggleButton.setOnMouseEntered(mouseEvent -> {
    Color red = Color.rgb(255, 0, 0, .90);
    pauseBackground.setStroke(red);
    firstLine.setStroke(red);
    secondLine.setStroke(red);
    playButton.setStroke(red);
});
playPauseToggleButton.setOnMouseExited(mouseEvent -> {
    Color white = Color.rgb(255, 255, 255, .90);
    pauseBackground.setStroke(white);
    firstLine.setStroke(white);
    secondLine.setStroke(white);
    playButton.setStroke(white);
});
// Boolean property toggling the playing and pausing
// the media player
playAndPauseToggle.addListener(
    (observable, oldValue, newValue) -> {

    if (newValue) {
        // Play
        if (mediaPlayer != null) {
            updatePlayAndPauseButtons(false, root);
            mediaPlayer.play();
        }
    } else {
        // Pause
        if (mediaPlayer!=null) {
            updatePlayAndPauseButtons(true, root);
            if (mediaPlayer.getStatus() == Status.PLAYING) {
                mediaPlayer.pause();
            }
        }
    }
});

// Press toggle button
playPauseToggleButton.setOnMouseClicked( mouseEvent ->{
    if (mouseEvent.getClickCount() == 1) {
        playAndPauseToggle.set(!playAndPauseToggle.get());
    }
});
```

```

buttonPanel.getChildren()
    .addAll(stopButton,
           playPauseToggleButton);
buttonPanel.setPrefWidth(50);

// Filter out to prevent the root node to
// receive mouse events to drag the window around.
buttonPanel.addEventHandler(MouseEvent.ANY, mouseEventConsumer);

return buttonPanel;
}

/**
 * The close button to exit application
 *
 * @return Node representing a close button.
 */
private Node createCloseButton() {

    StackPane closeButton = new StackPane();
    closeButton.setId("close-button");

    Node closeBackground = new Circle(0, 0, 7);
    closeBackground.setId("close-circle");
    SVGPath closeXmark = new SVGPath();
    closeXmark.setId("close-x-mark");
    closeXmark.setContent("M 0 0 L 6 6 M 6 0 L 0 6");

    closeButton.getChildren()
        .addAll(closeBackground,
               closeXmark);

    // exit app
    closeButton.setOnMouseClicked(mouseEvent -> {
        if (mediaPlayer != null){
            mediaPlayer.stop();
        }
        Platform.exit();
    });
    // Filter mouse events from propagating to the parent.
    closeButton.addEventHandler(MouseEvent.ANY, mouseEventConsumer);
    return closeButton;
}

/**
 * After a file is dragged onto the application a new MediaPlayer
 * instance is created with a media file.
 *
 * @param url - The URL pointing to an audio file.
 * @param root - The scene graph's root pane.
 */
private void playMedia(String url, Pane root) {

```

```

if (mediaPlayer != null) {
    mediaPlayer.pause();
    mediaPlayer.setOnPaused(null);
    mediaPlayer.setOnPlaying(null);
    mediaPlayer.setOnReady(null);
    mediaPlayer.currentTimeProperty()
        .removeListener(progressListener);
    mediaPlayer.setAudioSpectrumListener(null);
}

// create a new media player
Media validMedia = null;
try {
    validMedia = new Media(url);
} catch (Exception e) {
    new Alert(Alert.AlertType.ERROR, e.getMessage(), ButtonType.OK).showAndWait()
        .filter(response -> response == ButtonType.OK)
        .ifPresent(response -> e.printStackTrace());
    return;
}
final Media media = validMedia;

mediaPlayer = new MediaPlayer(media);

// as the media is playing move the slider for progress
mediaPlayer.currentTimeProperty()
    .addListener(progressListener);

mediaPlayer.setOnReady(() -> {
    // display media's metadata
    media.getMetadata().forEach( (name, val) -> {
        System.out.println(name + ": " + val);
    });
    updatePlayAndPauseButtons(false, root);
    Slider progressSlider =
        (Slider) root.lookup("#seek-position-slider");
    progressSlider.setValue(0);
    progressSlider.setMax(mediaPlayer.getMedia()
        .getDuration()
        .toSeconds());
    mediaPlayer.play();
});

// Rewind back to the beginning
mediaPlayer.setOnEndOfMedia( ()-> {
    updatePlayAndPauseButtons(true, root);
    // change buttons to the play button
    mediaPlayer.stop();
    playAndPauseToggle.set(false);
});

```

```

// Obtain chart path area
Path chartArea = (Path) root.lookup("#chart-area");

int chartPadding = 5;

// The frequency domain is the X Axis.
// The freqAxisY is the Y coordinate of the freq axis.
double freqAxisY = root.getHeight() - 45;

// The chart's height
double chartHeight = freqAxisY - chartPadding;

// In the CSS file the padding is set with the following:
//   -fx-border-insets: 6 6 6 6; (top, right, bottom, left)
//   -fx-border-width: 1.5;
// Below the padding will equal 7.5 which is the union of
// the inset 6 (left) and border width of 1.5)
double padding = root.getBorder().getInsets().getLeft();
double chartWidth = root.getWidth() - ((2 * padding) + (2 * chartPadding));

// Audio sound is in decibels and the magnitude is from 0 to -60
// Squaring the magnitudes stretches the plot. Also dividing into
// the chart height will normalize or keep the y coordinate within the
// chart bounds.
double scaleY = chartHeight / (60 * 60);
double space = 5; // between each data point. Helps stretch the chart width-wise.

mediaPlayer.setAudioSpectrumListener(
    (double timestamp,
     double duration,
     float[] magnitudes,
     float[] phases) -> {
        if (mediaPlayer.getStatus() == Status.PAUSED || mediaPlayer.getStatus() ==
            Status.STOPPED) {
            return;
        }
        // The freqBarX is the x coordinate to plot
        double freqBarX = padding + chartPadding;

        // The scaleX is one unit. This keeps the plotting within the chart width area
        double scaleX = chartWidth / (magnitudes.length * space);

        // Checks if the data array is created.
        // If not create the number of path components to be
        // added to the chartArea path. The check of the size minus 3 is excluding
        // the first MoveTo element, and the last two elements LineTo, and ClosePath
        // respectively.
        if ((chartArea.getElements().size() - 3) != magnitudes.length) {

```

```

// Move to bottom left of chart.
chartArea.getElements().clear();
chartArea.getElements().add(new MoveTo(freqBarX, freqAxisY));

// Update all LineTo elements to draw the line chart
for(float magnitude:magnitudes) {
    double dB = magnitude * magnitude;
    dB = chartHeight - dB * scaleY;
    chartArea.getElements().add(new LineTo(freqBarX, freqAxisY - dB));
    freqBarX+=(scaleX * space);
}

// Close the path by adding LineTo to the bottom right
// of the chart and close path to form an shape.
chartArea.getElements().add(new LineTo(freqBarX, freqAxisY));
chartArea.getElements().add(new ClosePath());
} else {
    // If elements already created
    // go through and update path elements
    int idx = 0;
    for(float magnitude:magnitudes) {
        double dB = magnitude * magnitude;
        dB = chartHeight - dB * scaleY;

        // skip first MoveTo element in path.
        idx++;

        // update elements with a x and y
        LineTo dataPoint = (LineTo) chartArea.getElements().get(idx);
        dataPoint.setX(freqBarX);
        dataPoint.setY(freqAxisY - dB);
        freqBarX += (scaleX * space);
    }
}
});

}

/**
* Sets play button visible and pause button not visible when
* playVisible is true otherwise the opposite.
*
* @param playVisible - value of true the play becomes visible
* and pause non visible, otherwise the opposite.
* @param root - The root node (AnchorPane)
*/
private void updatePlayAndPauseButtons(boolean playVisible, Parent root) {
    Node playButton = root.lookup("#play-button");
    Node pauseButton = root.lookup("#pause-button");
}

```

```

if (playVisible) {
    // show play button
    playButton.toFront();
    playButton.setVisible(true);
    pauseButton.setVisible(false);
    pauseButton.toBack();
}

} else {
    // show pause button
    pauseButton.toFront();
    pauseButton.setVisible(true);
    playButton.setVisible(false);
    playButton.toBack();
}

}

/** 
 * A position slider to seek backward and forward
 * that is bound to a media player control.
 *
 * @return Slider control bound to media player.
 */
private Slider createSlider() {
    Slider slider = new Slider(0, 100, 1);
    slider.setId("seek-position-slider");
    slider.valueProperty()
        .addListener((observable) -> {
            if (slider.isValueChanging()) {
                // must check if media is paused before seeking
                if (mediaPlayer != null &&
                    mediaPlayer.getStatus() == MediaPlayer.Status.PAUSED) {

                    // convert seconds to millis
                    double dur = slider.getValue() * 1000;
                    mediaPlayer.seek(Duration.millis(dur));
                }
            }
        });
    return slider;
}
}

```

The application's JavaFX CSS styling file named `playing-audio.css`, as shown in Listing 9-2, contains IDs and classes for various elements in the audio player.

Listing 9-2. The JavaFX CSS Styling for the Audio Player Application

```

.root {
    -white-ish: rgba(255, 255, 255, .90);
    -black-ish: rgba(0, 0, 0, .80);
}

```

```
#app-surface {  
    /* Application surface */  
    -fx-snap-to-pixel: false;  
  
    /* Background */  
    -fx-background-color: -black-ish;  
    -fx-background-radius: 7;  
    -fx-background-insets: 2 2 2 2;  
  
    /* Border */  
    -fx-border-color: -white-ish;  
    -fx-border-style: solid;  
    -fx-border-insets: 6 6 6 6;  
    -fx-border-width: 1.5;  
    -fx-border-radius: 5;  
}  
  
#chart-area {  
    -fx-stroke: white;  
    -fx-stroke-width: 1.5;  
    -fx-stroke-type: outside;  
    -fx-smooth: true;  
    -fx-stroke-line-cap: round;  
    -fx-stroke-line-join: round;  
    /*  
        chartPadding = 5  
        freqAxisY = 225  
    */  
    -fx-fill: linear-gradient(from 5 5 to 5 225,  
        rgba(255, 0, 0, .70) 0%,      /* Red      */  
        rgba(255, 165, 0, .70) 40%,   /* Orange   */  
        rgba(255, 255, 0, .70) 70%,   /* Yellow   */  
        rgba(0, 255, 0, .70) 80%,    /* Green    */  
        rgba(0, 0, 255, .70) 90%,    /* Blue     */  
        rgba(75,0,128,.70) 95%,     /* Indigo   */  
        rgba(238,130,238,.70) 100%) /* Violet  */  
}  
  
#button-panel {  
    /* Background */  
    -fx-background-color: -black-ish;  
    -fx-background-radius: 7;  
  
    /* Border */  
    -fx-border-radius: 5;  
    -fx-border-color: -white-ish;  
    -fx-border-style: solid;  
    -fx-border-width: 1.5;  
    -fx-stroke: -white-ish;  
    -fx-alignment: center-right;  
    -fx-hgap: 5;  
}
```

```
#stop-button {  
    -fx-arc-width: 5;  
    -fx-arc-height: 5;  
    -fx-fill: -white-ish;  
    -fx-stroke: -white-ish;  
}  
#stop-button:hover {  
    -fx-stroke: red;  
}  
  
#play-button {  
    -fx-fill: -white-ish;  
}  
#play-button:hover {  
    -fx-stroke: derive(red, 90%);  
}  
  
.pause-circle {  
    -fx-stroke: -white-ish;  
    -fx-stroke-width: 1.5;  
}  
  
.pause-line {  
    -fx-stroke-width: 3;  
    -fx-stroke: -white-ish;  
    -fx-translate-y: 6;  
}  
  
.pause-line.first-line {  
    -fx-translate-x: 4;  
}  
  
.pause-line.second-line {  
    -fx-translate-x: 8;  
}  
  
/* Close button */  
#close-x-mark {  
    -fx-stroke-width: 1.5;  
    -fx-stroke: -black-ish;  
}  
  
#close-circle {  
    -fx-stroke: rgba(235, 90, 78, .90);  
    -fx-fill: rgba(186, 90, 78, .90);  
    -fx-snap-to-pixel: false;  
}  
#close-circle:hover {  
    -fx-stroke: derive(red, 90%);  
}
```

```

/* Play progress and seek slider */
.slider {
    -fx-show-tick-labels: false;
    -fx-show-tick-marks: true;
}

/* A triangle shaped thumb */
.slider .thumb {
    -fx-shape: "M 2 2 L 6 6 L 2 10 L 3 6 Z";
}
.slider .thumb:hover {
    -fx-border-color:derive(red, 90%);
}
.slider .track {
    -fx-pref-height: 6;
    -fx-background-radius: 2;
    -fx-background-color: derive(black, 40%);
    -fx-border-color:derive(gray, 80%);
}
.slider .axis .axis-label {
    -fx-text-fill: white;
}

```

How It Works

Before beginning, I want to describe some changes that I made in this version of the book from the earlier edition. I basically refactored (yet again) the source code from the prior book's edition relating to media. Because of this refactoring, I mention privately scoped methods quite often. These private methods are used to break up the code into more manageable pieces, so that you can better focus on each feature of the media player application.

Assuming you've built and ran Listing 9-1, you will begin by dragging and dropping an audio file onto the example MP3 player, as you did in Chapter 7's Photo Viewer application, where you learned how to use the drag-and-drop desktop metaphor to load files into a JavaFX application. Instead of image files, however, you load audio files. To load audio files, JavaFX currently supports the following file formats: `.mp3`, `.wav`, `.aiff`, and `.m3u8`. To see the supported encoding types, refer to the Javadocs documentation of the package summary of `javafx.scene.media`.

Looking back at Figure 9-2, you might notice that the button panel control has a look and feel like the Photo Viewer application in Figure 7-1 of Chapter 7. In this example, I've modified the button controls to resemble buttons, similar to many media player applications.

As an bonus, the MP3 player will appear as an irregularly shaped, semitransparent window without a title bar or borders. The application window can also be dragged around the desktop using the mouse. Now that you know how to operate the music player, let's walk through the code.

The Audio Player Application's Instance Variables

First, the code contains instance variables at the top of the class that will maintain state information for the lifetime of the application. Table 9-2 describes all instance variables used in the MP3 audio player application.

Table 9-2. MP3 Player Application Instance Variables

Variable	Data Type	Example	Description
mediaPlayer	MediaPlayer	-	A media player reference that plays audio and video.
anchorPt	Point2D	(100,100)	Screen coordinates where the user begins to drag the stage window.
previousLocation	Point2D	(0, 0)	The upper-left corner of the stage's previous coordinate; assists in dragging the stage window.
progressListener	ChangeListener<Duration>	5 (sec.)	Bound to the media player's <code>currentTimeProperty()</code> method. Updates the UI slider control.
playAndPauseToggle	BooleanProperty	true	A Boolean property having a change listener to pause and play an audio file. When the Pause button is displayed, the property is true; otherwise, it is false.
mouseEventConsumer	EventHandler<MouseEvent>	-	Event filter to prevent mouse events sent to the parent node.

In Table 9-2, the first variable is a reference to a media player (`MediaPlayer`) object that will be created in conjunction with a `Media` object containing an audio file. Next, you can see two variables named `anchorPt` and `previousLocation`. These variables are used to maintain the stage window's positions on the desktop when the user drags the application window across the screen.

The `anchorPt` variable is used to save the starting coordinates of a mouse press when the user begins to drag the window. When calculating the upper-left bounds of the application window during a mouse-drag operation, the `previousLocation` variable will contain the previous window's screen x- and y-coordinates.

After describing the variables that keep track of the window's position, you will see the `progressListener` variable. Here the `progressListener` is responsible for adjusting the slider's indicator (thumb) as the audio file is being played.

The variable `playAndPauseToggle` is responsible for determining when the user has clicked on the Pause and Play button controls. As you will see later, the property will have an attached change listener to show and hide the Play and Pause buttons inside of a `StackPane` (see `updatePlayAndPauseButtons()`).

Lastly, the variable `mouseEventConsumer` is an event filter to be applied to the button panel and Close button. Event filters will prevent mouse events from propagating to the parent node (`AnchorPane`). In this scenario, the parent node is the root node of the Scene graph. This prevents dragging the window when the mouse cursor is clicking on the button panel and Close button.

While the application could've had more state variables to reference objects such as nodes on the Scene graph, you can actually look them up by ID. This of course assumes that a JavaFX node's unique ID was set using the `setId()` method. For example, if the Scene graph contains a node with an ID of `stop-button` (`node.setId("stop-button")`), you could later invoke the `scene.lookup("#stop-button")` method to retrieve the Stop button node from the Scene graph. Remember to prefix the id with an # symbol.

Setting Up the Stage Window

Now that you know about the instance (state) variables, let's jump into the `start()` method code block. In previous chapters relating to GUIs, you saw that GUI applications normally contain a title bar and windowed borders surrounding the scene. Here, I want to demonstrate the ability to create irregularly shaped semi-transparent windows. The code invokes the stage (`primaryStage`) object's `initStyle()` method with a `StageStyle.TRANSPARENT` value; as a result, the stage window has no title bar (is *undecorated*) and is transparent.

After initializing the stage's style, the code creates a root node as an `AnchorPane`. I use an `AnchorPane` because it is easy to position the controls relative to the edges of the application window. You'll notice the `setId()` method called with `app-surface` as its unique ID. If you are not familiar with JavaFX CSS, spend some time looking at Chapter 15 on topic of custom UIs. We will employ the styling definitions shown in Listing 9-2. To reiterate the portion of CSS styling code, the root node from Listing 9-2 is shown here:

```
.root {
    -white-ish: rgba(255, 255, 255, .90);
    -black-ish: rgba(0, 0, 0, .80);
}

#app-surface {
    /* Application surface */
    -fx-snap-to-pixel: false;

    /* Background */
    -fx-background-color: -black-ish;
    -fx-background-radius: 7;
    -fx-background-insets: 2 2 2 2;

    /* Border */
    -fx-border-color: -white-ish;
    -fx-border-style: solid;
    -fx-border-insets: 6 6 6 6;
    -fx-border-width: 1.5;
    -fx-border-radius: 5;
}
//... Code omitted
```

After applying the CSS styling to the root node, as shown in Figure 9-5, you'll notice a translucent rounded background with a white border inset slightly from the outer edges.



Figure 9-5. JavaFX CSS styling the root node of the Audio Player application. The CSS selector `app-surface` will create a translucent rounded background with a white border inset slightly from the outer edges.

Next, the code takes the usual step of setting up the scene by applying a JavaFX style sheet file `playing-audio.css`. Listing 9-2 shows the contents of the `playing-audio.css` file. Notice that the `root` class selector has two global color attributes, named `-white-ish` and `-black-ish`. I created these to be reused in other styling definition blocks. If you are not familiar with JavaFX CSS styling, refer to Chapter 15 on custom UIs.

Back in the `start()` method from Listing 9-1, after applying the CSS style sheet, the code invokes the `private initMovablePlayer()` method, which is responsible for making the stage window dragable. Because you don't have a native title bar, you will find it convenient to be able to drag any part of the application's surface. The `initMovablePlayer()` method uses the instance variables `anchorPt` and `previousLocation` mentioned earlier to calculate points based on relative screen coordinates when mouse events (such as press, drag, or release) are triggered.

The Spectrum Area Chart Visualization (the `Scene.lookup()` Method)

After creating a scene and stage capable of being dragged using the mouse, the code creates a `Path` instance with an ID of `chart-area`. This path will contain `Path` elements that will draw a spectrum chart as a visualization as the audio media is being played. As you will see later in the code, the `MediaPlayer` instance will attach an audio spectrum listener to update path elements that will draw an area chart filled with a linear gradient.

While a JavaFX node having a unique ID can be styled using CSS selectors, the ID can also be used to be recalled or looked up within a parent node or a scene by using the `lookup(id)` method. Be aware that an ID set on a node should be unique. According to the Javadocs documentation, “If more than one node matches the specified selector, this function returns the first of them. If no nodes are found with this ID, then null is returned.”

Note To look up a node in a Scene graph or a parent node, use the `lookup()` method by passing in the child's ID. Be sure to prefix the string ID with the symbol `#`. For example, `Node myButton = scene.lookup("#my-button");`

Using the scene's `lookup()` method is a convenient way to retrieve nodes from the Scene graph in a random-access way without having to create globally defined instance variables.

Creating a Custom Button Panel

Continuing with the `start()` method, the code creates a *custom button panel* by invoking the private method `createButtonPanel()`. After invoking the `createButtonPanel()` method to create a custom button panel, the code will position the control to the bottom-right corner. The following code snippet sets the button panel to the bottom-right corner with a 3.0 pixel padding.

```
AnchorPane.setRightAnchor(buttonPanel, 3.0);
AnchorPane.setBottomAnchor(buttonPanel, 3.0);
```

Looking deeper into the `createButtonPanel()` method, it begins by using JavaFX shapes that will represent the Stop, Play, and Pause buttons for the MP3 audio player. When creating shapes or custom nodes, you can add event handlers to nodes in order to respond to mouse events. Although there are advanced ways to build custom controls in JavaFX, I chose to build my own button icons from simple rectangles, arcs, circles, and lines. To see more advanced ways to create custom controls, refer to Chapter 15 on custom UIs.

To attach event handlers to a mouse press, simply call the `setOnMouseClicked()` method by passing in an `EventHandler<MouseEvent>` instance. The following code block adds a lambda expression that responds to a mouse click for the `stopButton` node. Listing 9-3 shows the code for the Stop button from the private method `createButtonPanel()` from Listing 9-1. The Stop button will toggle the Pause button to a Play button and position the player to the beginning of the file. The event handler for a mouse clicked event is attached to the Stop button.

Listing 9-3. Code for the Stop Button from the Private Method `createButtonPanel()`

```
// stop button control
Node stopButton = new Rectangle(10, 10);
stopButton.setId("stop-button");
stopButton.setOnMouseClicked(mouseEvent -> {
    if (mediaPlayer != null) {
        updatePlayAndPauseButtons(true, root);
        if (mediaPlayer.getStatus() == Status.PLAYING ||
            mediaPlayer.getStatus() == Status.PAUSED) {

            mediaPlayer.stop();
        }
        playAndPauseToggle.set(false);
    }
});
```

In Listing 9-3, a square node (`Rectangle`) is defined as the Stop button, set with an ID of `stop-button`. The ID will uniquely identify a node for the CSS styling in Listing 9-2. Next, the code sets an event handler for the `OnMouseClicked` event to stop the media player from playing. You'll notice the method call to `updatePlayAndPauseButtons(true, root)`, which toggles the display of the Play button and hides the Pause button. Otherwise, you pass in a Boolean value of `false`, so the Pause button will be displayed and the Play button will be hidden. The last parameter is the root node to be used to look up the Play and Pause buttons.

Still within the `createButtonPanel()` method, the rest of the code creates a `StackPane` to contain the Pause and Play button shapes. Next, you see that the Play button is created using an Arc shape. When creating a toggle-like button for the Pause and Play controls, the code uses a `StackPane` as a container to hold both Pause and Play shapes. Since I wanted to treat the `StackPane` as one button, I had to ignore the mouse events on the Pause and Play shapes. Both Play and Pause nodes ignore mouse events by invoking the `setMouseTransparent(true)` method. That way, the `OnMouseClicked` event's handler code is attached to the `StackPane` to toggle which button shape to display and to update the Boolean `playAndPauseToggle` property.

The Boolean property variable `playAndPauseToggle` is set to `false`. The `playAndPauseToggle` property is responsible for actually pausing and playing the media player. Near the end of the `createButtonPanel()` method, a change listener is attached to the `playAndPauseToggle` property, which responds when the value changes. The code shown here is from Listing 9-2, but it's reshown in Listing 9-4 to point out that the property is used to actually control the pausing and playing the media player. When the Boolean property is `true` the Pause button is shown and the media gets played. Otherwise, the Play button is shown and the media is paused.

Listing 9-4. Change Listener Added to the Boolean Property `playAndPauseToggle`

```
// Boolean property toggling the playing and pausing
// the media player
playAndPauseToggle.addListener((observable, oldValue, newValue) -> {
    if (newValue) {
        // Play
        if (mediaPlayer != null) {
            updatePlayAndPauseButtons(false, root);
            mediaPlayer.play();
        }
    } else {
        // Pause
        if (mediaPlayer!=null) {
            updatePlayAndPauseButtons(true, root);
            if (mediaPlayer.getStatus() == Status.PLAYING) {
                mediaPlayer.pause();
            }
        }
    }
});
```

The media player object is a state machine that transitions between states internally. Some methods use other threads that aren't on the JavaFX application thread. So it is typically good practice to query the state of the media player before invoking methods such as the `pause()` or `stop()` method. In the case of the Pause button, you've seen that the code checks for the `Status.PLAYING` before invoking the `pause()` method on the media player object.

Because all the buttons use code similar to Listing 9-3 and the preceding code snippet (`EventHandler`), I will list only the method calls that each button will ultimately perform on the media player. The following actions are responsible for stopping, pausing, playing, and exiting the MP3 player application:

```
Stop - mediaPlayer.stop();
Pause - mediaPlayer.pause();
Play - mediaPlayer.play();
Close - Platform.exit();
```

Lastly, before returning the button panel, the code adds an event filter to prevent any mouse events to be received on its parent node. Because the parent node (`root`) has code that listens for mouse drag events, you don't want it to interfere when the user is trying to click on buttons inside the button panel. In other words, you can't drag the application around the screen when your mouse pointer is over the button panel region. The following statement filters any mouse events that are propagated to the parent node.

```
// Filter out to prevent the root node to
// receive mouse events to drag the window around.
buttonPanel.addEventFilter(MouseEvent.ANY, mouseEventConsumer);
```

Play Progress, Rewind, and Fast Forward

After creating the custom button panel in the `start()` method, the code invokes the private method `createSlider()`, which returns a JavaFX Slider UI control. The slider control represents a progress position slider that moves based on the position of the media as it is being played. When the media is paused, the user may also drag the slider control's "thumb" to go backward and forward in time through the audio or video media.

Seeking Backward or Forward in the Media

In the private method `createSlider()`, the following code implements a slider control representing a progress (seek) position slider control for the MP3 audio player application:

```
Slider slider = new Slider(0, 100, 1);
slider.setId("seek-position-slider");
slider.valueProperty()
    .addListener((observable) -> {
        if (slider.isValueChanging())
            // must check if media is paused before seeking
            if (mediaPlayer != null &&
                mediaPlayer.getStatus() == MediaPlayer.Status.PAUSED) {

                // convert seconds to millis
                double dur = slider.getValue() * 1000;
                mediaPlayer.seek(Duration.millis(dur));
            }
    });
});
```

Notice the slider control's `valueProperty()` method and the call to the `addListener()` method, which has an `InvalidationListener` (lambda expression) that detects whether the user has moved the slider control, thus changing the value property. The lambda parameter for an `InvalidationListener` (functional interface) is an object of type `javafx.beans.Observeable` that refers to the slider's value property.

The code first checks whether the slider's value has changed by using the method `isValueChanging()`. It's important to perform this check before the `seek()` method because of performance efficiencies. Using the `InvalidationListener`, the slider's value changes only when it's needed (an example of lazy evaluation). If you used a `ChangeListener` (eager evaluation), the value would change too often and could overwhelm the media player. As a reminder, the `InvalidationListener` is a lambda with a single parameter of type `Observeable`, while a `Changelistener` is a lambda expression with three parameters consisting of an `ObserveableValue`, an old value, and a new value.

Using a Slider Control to Show Play Progress

Still in the `start()` method, the code assigns the variable `progressListener` to a `ChangeListener` (lambda) instance, as shown in the following code snippet:

```
// Updates slider as audio/video is progressing (play)
progressListener = (observable, oldValue, newValue) ->
    progressSlider.setValue(newValue.toSeconds());
```

The `progressListener` is responsible for updating the slider's "thumb" as the media player is playing audio or video media. The listener is added to the media player object's `currentTimeProperty()` in the `playMedia()` method. In the private method `playMedia()`, the code first cleans up by removing the singleton `progressListener` instance, which was bound to an old media player object, and adds the `progressListener` onto a newly created media player object. By removing old references to the progress listener, you allow the Java runtime to perform its garbage collection process, thus preventing memory leaks. Currently, a media object cannot be replaced in a `MediaPlayer` instance with a setter. In order to load new media, a new `MediaPlayer` instance must be created. You'll notice in the code showing the media loaded using the constructor `MediaPlayer(media)` that it takes a `Media` instance.

Setting Up Drag-and-Drop Support for Audio Files

Continuing with the `start()` method, notice the call to a private method `initFileDragNDrop()`. I created this method to support the drag-and-drop capability that is responsible for loading an audio file from the local filesystem or browser URL address field. Because the implementation is nearly identical to Chapter 5's Photo Viewer example, I will not discuss the code details, except to mention the invocation of the private method `playMedia()`. After the file is dragged and dropped, the URL string is passed into the method `playMedia()`. This method will conveniently clean up and create a new media player and immediately play the media file.

Audio File Drag and Drop

```
/**
 * Initialize the Drag and Drop ability for media files.
 * @param root - The Scene graph's root pane.
 */
private void initFileDragNDrop(Pane root)
```

Inside the `initFileDragNDrop()` method is the `OnDragDropped` event handler code, which obtains the file from the `DragEvent`'s drag board. Here, the URL path is extracted and passed in to the private `playMedia()` method. The `playMedia()` method begins by checking for the existence of a previously created `mediaPlayer` object. If there was a previous media player instance, the code would stop the media player and clean up any old references to events, listeners, and properties.

Media Metadata

Next, the code creates a `Media` class instance with a valid URL string. After creating a `Media` instance, the code loops through the media's metadata via the `getMetadata()` method to display the keys as output to the console. Sometimes MP3 music files will contain information about the artist, tracks, or song titles. By interrogating the media object's metadata, you can display the information to the user.

After listing the media's metadata, the code instantiates a new `MediaPlayer` instance with a new `Media` object and adds the `progressListener` onto the media player's `currentTimeProperty()` method. As mentioned earlier for the progress position slider control, the "thumb" will progress forward as the media is being played.

Playing Media

```
private void playMedia(String url, Pane root)
```

After a file is dragged on the surface to be played, the code passes the URL and root Pane to the private `playMedia()` method. The code does some cleanup if there was a prior media player by pausing and clearing any event handlers or listeners attached.

Next, the code will load validate if the media is supported. When loading a media file using the `Media` class constructor, you could get a `MediaException` that's likely a media unsupported issue. The code will show a JavaFX Alert window to the user. Listing 9-5 shows how to validate if the media file is supported or not. If not, a JavaFX Alert dialog window appears to the users.

Listing 9-5. Code to Determine If the Media File Is Supported.

```
// create a new media player
Media validMedia = null;
try {
    validMedia = new Media(url);
} catch (Exception e) {
    new Alert(Alert.AlertType.ERROR,
        e.getMessage(),
        ButtonType.OK)
        .showAndWait()
        .filter(response -> response == ButtonType.OK)
        .ifPresent(response -> e.printStackTrace());
    return;
}
final Media media = validMedia;
```

Continuing with the `playMedia()` method, the code instantiates a new `MediaPlayer` with the new media. Next, the `mediaPlayer` will attach the `progressListener` defined earlier to its `currentTimeProperty()` method.

```
mediaPlayer.setOnReady()
```

After setting up the media player object, the code is ready to play the file. The code passes in a created `Runnable` (lambda expression) instance to respond to the `OnReady` event on the media player object. The code first resets the Play and Pause buttons of the custom button panel by invoking the `updatePlayAndPauseButtons()` method. Passing in `true` displays the Play button and hides the Pause button; otherwise, the Pause button is shown and the Play button is hidden.

Next, the code initializes the slider control based on the media's total duration. Finally, the code invokes the media player object's `play()` method to begin playing the music media.

Rewinding (the `OnEndOfMedia` Event)

Continuing inside the `playMedia()` method, after setting up the media player's `OnReady` event, the code sets up the `OnEndOfMedia` event. This event is triggered when the media player play position has reached the end. The code toggles the Play and Pause buttons' appearance and rewinds the media to the start position of the file, also placing it in a Ready state.

Updating the Visualization Using the AudioSpectrumListener Interface

Do you remember the earlier discussion about the MP3 audio player's visualization of a spectrum area chart dancing about as the music plays? Still within the `playMedia()` method, let's look at how to implement this mesmerizing effect.

Continuing with the `playMedia()` method, the code will obtain the `Path` node having the ID `chart-area` that was created earlier. The following line retrieves the `Path` node using the root node's `lookup()` method:

```
// Obtain chart path area
Path chartArea = (Path) root.lookup("#chart-area");
```

Next, the code sets the media player's audio spectrum listener with an `AudioSpectrumListener` (lambda expression) instance. The listener is notified periodically with updates of the audio spectrum. The following code is shown again from Listing 9-1. It shows the code that updates the `Path` node (`chart-area`) with a linear gradient color based on the magnitude array. Listing 9-6 shows the area chart plotted using the audio spectrum data. Every interval the audio spectrum data will plot an area chart. Based on the root pane's size and defined chart padding, the code will use the magnitude array data to draw a closed path to be filled with a linear gradient.

Listing 9-6. Area Chart Plotted Using the Audio Spectrum Data

```
int chartPadding = 5;

// The frequency domain is the X Axis.
// The freqAxisY is the Y coordinate of the freq axis.
double freqAxisY = root.getHeight() - 45;

// The chart's height
double chartHeight = freqAxisY - chartPadding;

// In the CSS file the padding is set with the following:
//   -fx-border-insets: 6 6 6 6; (top, right, bottom, left)
//   -fx-border-width: 1.5;
// Below the padding will equal 7.5 which is the union of
// the inset 6 (left) and border width of 1.5)
double padding = root.getBorder().getInsets().getLeft();
double chartWidth = root.getWidth() - ((2 * padding) + (2 * chartPadding));

// Audio sound is in decibels and the magnitude is from 0 to -60
// Squaring the magnitudes stretches the plot. Also dividing into
// the chart height will normalize or keep the y coordinate within the
// chart bounds.
double scaleY = chartHeight / (60 * 60);
double space = 5; // between each data point. Helps stretch the chart width-wise.

mediaPlayer.setAudioSpectrumListener(
    (double timestamp,
     double duration,
     float[] magnitudes,
     float[] phases) -> {
```

```

if (mediaPlayer.getStatus() == Status.PAUSED || mediaPlayer.getStatus() == Status.STOPPED) {
    return;
}
// The freqBarX is the x coordinate to plot
double freqBarX = padding + chartPadding;

// The scaleX is one unit. This keeps the plotting within the chart width area.
double scaleX = chartWidth / (magnitudes.length * space);

// Checks if the data array is created.
// If not create the number of path components to be
// added to the chartArea path. The check of the size minus 3 is excluding
// the first MoveTo element, and the last two elements LineTo, and ClosePath
// respectively.
if ((chartArea.getElements().size() - 3) != magnitudes.length) {

    // Move to bottom left of chart.
    chartArea.getElements().clear();
    chartArea.getElements().add(new MoveTo(freqBarX, freqAxisY));

    // Update all LineTo elements to draw the line chart
    for(float magnitude:magnitudes) {
        double dB = magnitude * magnitude;
        dB = chartHeight - dB * scaleY;
        chartArea.getElements().add(new LineTo(freqBarX, freqAxisY - dB));
        freqBarX+=(scaleX * space);
    }

    // Close the path by adding LineTo to the bottom right
    // of the chart and close path to form an shape.
    chartArea.getElements().add(new LineTo(freqBarX, freqAxisY));
    chartArea.getElements().add(new ClosePath());
} else {
    // If elements already created
    // go through and update path elements
    int idx = 0;
    for(float magnitude:magnitudes) {
        double dB = magnitude * magnitude;
        dB = chartHeight - dB * scaleY;

        // skip first MoveTo element in path.
        idx++;

        // update elements with a x and y
        LineTo dataPoint = (LineTo) chartArea.getElements().get(idx);
        dataPoint.setX(freqBarX);
        dataPoint.setY(freqAxisY - dB);
        freqBarX += (scaleX * space);
    }
}
});
```

Listing 9-6 creates an area chart based on `Path` elements to denote the magnitude at a particular frequency. Beginning with the setup of the chart, the code defines padding of five pixels after the left inset of the root pane. Next, the `freqAxisY` variable is declared to contain the y-coordinate to draw the frequency axis (*x-axis*). Next, the `chartHeight` and `chartWidth` variables are defined and calculated. Because known insets and border widths are defined in CSS, the code will invoke the root pane's `getBorder().getInsets().getLeft()` to obtain the left padding calculation. Padding is the union of the inset and border widths.

In Figure 9-6, the `freqAxisY` variable is the y-screen coordinate (fixed value) of the bottom of the area chart representing the frequency domain on the x-axis. The `dB` is a calculated value representing the decibels at a particular frequency for plotting the y-coordinate of each magnitude (data point).

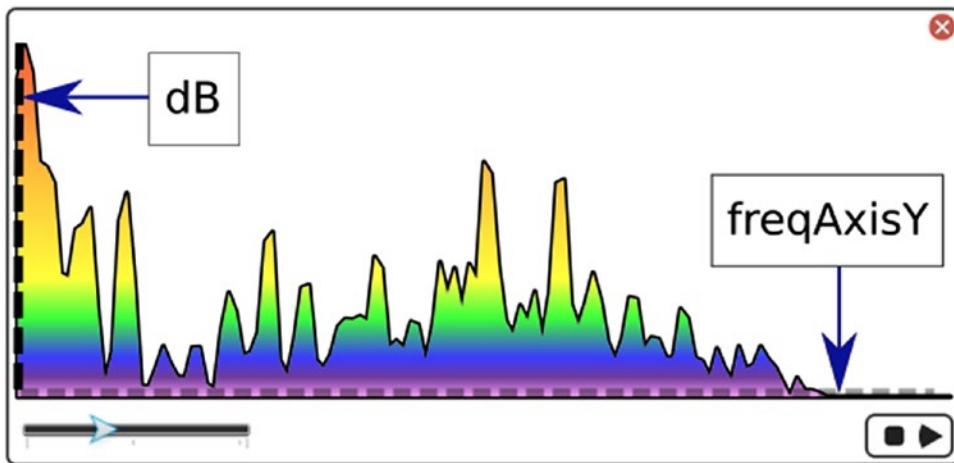


Figure 9-6. The area chart's x-axis (`freqAxisY`) and y-axis (`dB`)

After the area chart's position, the width and height are calculated as a `scaleY` variable is needed to help normalize or to keep the plotting points in the application window bounds. The y-axis represents the decibels where some magnitude values can be plotted off (the chart) of the display surface. The last variable defined is the `space` variable, and it will be used to represent the number of pixels between each data point to be plotted.

Now for the fun part—creating the spectrum area chart visualization. The media player can get notifications of audio spectrum data on regular intervals. By default, the updates are emitted every 0.1 seconds (100ms). Of course you can change the interval by calling the media player's `setAudioSpectrumInterval(double)` method. To respond to audio spectrum notification updates, the code invokes the media player's `setAudioSpectrumListener()`. The method signature is shown here:

```
mediaPlayer.setAudioSpectrumListener(
    (double timestamp,
     double duration,
     float[] magnitudes,
     float[] phases)
```

According to the Javadocs, the listener is an observer receiving periodic updates of the audio spectrum. In layman's terms, it is the audio media's sound data such as volume, tempo, and so on. To create an instance of an `AudioSpectrumListener`, you will create a lambda expression with parameters of the interface's method `spectrumDataUpdate()`. Table 9-3 lists all the inbound parameters for the audio spectrum listener's method. For more details, refer to the Javadocs documentation of `javafx.scene.media.AudioSpectrumListener`.

Table 9-3. The *AudioSpectrumListener*'s Method *spectrumDataUpdate()* Inbound Parameters

Parameters	Data Type	Example	Description
timestamp	Double	2.4261	When the event occurred, in seconds.
duration	Double	0.1	The duration of time (in seconds) the spectrum was computed.
magnitudes	float[]	-50.474	An array of float values representing each band's spectrum magnitude in decibels (non-positive float value).
phases	float[]	1.22	An array of float values representing each band's phase. Each value range is between -Math.PI and +Math.PI.

The code for the listener begins by checking if the media player's current state is Paused or Stopped. If so, it returns immediately. If the state is neither Paused nor Stopped, it's time to draw the spectrum area chart. The code starts off declaring and pre-computing variables to contain the data points' beginning x-coordinate value. The *scaleX* variable is used to scale points to ensure that they are kept within the chart bounds width-wise. Next is the initialization of the chart's Path node object. The chart's Path node will get initialized with Path elements to form a closed path based on the array of magnitudes passed in.

When creating the shape of the area chart, the first Path element will be a MoveTo. The last two Path elements will be a LineTo and a ClosePath element. The last two Path elements are drawn from the last data point to the bottom-right of the chart (visible x-axis), then closing the path using the ClosePath element. The Path elements that are between the first and last elements will be updated every update interval. Because all magnitude values are negative numbers between 0 and -60, the squares of the values for the amplitude or peak of a data point. Since the chart is really a shape, it can be filled with a color. In this example, the code will fill the chart with a linear shape based on the color spectrum (ROYGBIV). The following code snippet is a JavaFX fill attribute assigned a linear gradient for the area chart shape.

```
-fx-fill: linear-gradient(from 5 5 to 5 225,
    rgba(255, 0, 0, .70) 0%, /* Red */
    rgba(255, 165, 0, .70) 40%, /* Orange */
    rgba(255, 255, 0, .70) 70%, /* Yellow */
    rgba(0, 255, 0, .70) 80%, /* Green */
    rgba(0, 0, 255, .70) 90%, /* Blue */
    rgba(75,0,128,.70) 95%, /* Indigo */
    rgba(238,130,238,.70) 100%) /* Violet */
```

Quitting (the Close Button)

```
private Node createCloseButton()
```

Lastly is the ability to quit the application in the *start()* method by creating a Close button by invoking the private method *createCloseButton()*, which returns a custom node that has handler code to exit the application. When creating the Close button, I created a stack pane containing a circle and an SVGNode. The circle and SVGNode is styled using CSS from Listing 9-2. There is nothing unusual about the styling except for the pseudo-class called *hover* when the mouse cursor is on top of the Close button to highlight the button. The following code snippet changes the opacity level of the stroke color when the mouse is hovering over the Close button (circle shape).

```
#close-circle:hover {
    -fx-stroke: derive(red, 90%);
}
```

The `createClose` button method's main job is to stop the media player and exit the application by calling `stop()` and `Platform.exit()`.

Playing Video

Playing videos in JavaFX is quite simple. You use the same APIs as with audio to load media into a JavaFX media player object. The only difference is that you need a `MediaView` node to display the video on the scene. Because `MediaView` is a JavaFX node, you can transform, scale, translate, and apply effects. In this section, you learn about cross-platform video media file formats, and later you see an example of a JavaFX video player.

As of the writing of this book, the JavaFX media player API supports the video formats MPEG-4 H.264/AVC (.mp4) and VP6 using an .flv container format.

MPEG-4

MPEG-4 is a multimedia container format containing H.264/AVC video. Currently MPEG-4 is a compressed video format that is widely used to view video on the Internet. You'll find that some high-definition video recorders can automatically convert raw video to MPEG-4 to produce files with the extension .mp4. For the devices that do not produce mp4 files, you currently need to purchase movie or video editing software capable of converting raw videos into MP4-formatted files.

The following software programs, among others, can convert raw video into the portable MPEG4 format:

- Adobe Premiere
- Apple Final Cut Pro
- Apples iMovie
- Apple Quick Time Pro
- Sony Vegas Movie Studio
- Sony Vegas Pro
- FFmpeg

VP6 .flv

JavaFX 2.x and later support a cross-platform video format called VP6 with a file extension of .flv (which stands for the popular Adobe Flash Video format). The actual encoder and decoder (codec) to create VP6 and .flv files are licensed through a company called On2. In 2010, On2 was acquired by Google to build VP7 and VP8 to be open and free to advance HTML5.

Refer to the Javadocs for more details on the formats to be used. A word to the wise: beware of web sites claiming to be able to convert videos for free. As of this writing, the only encoders capable of encoding video to VP6 legally are the commercial converters from Adobe.

You are probably wondering how to obtain a .flv file when you don't have encoding software. If you don't have an .flv file lying around, you can obtain one from one of my favorite sites, called Media College (<http://www.mediacollege.com>). From photography to movies, Media College provides forums, tutorials, and resources that help guide you into the world of media.

There you can obtain a particular media file to be used in one of the examples in this chapter relating to closed captioning. To obtain the .flv file, navigate to the following URL:

<http://www.mediacollege.com/adobe/flash/video/tutorial/example-flv.html>.

Next, locate the link entitled Windy 50s Mobility Scooter Race, which points to the .flv media file (20051210-w50s.flv). In order to download a link to the desired file, right-click to select Save Target As or Save. Once you have saved the file locally on your filesystem, you can use it for the examples in this chapter.

It is *not* required that you have .flv files for this chapter. You may use MPEG-4 (mp4) formatted files for the examples.

A Video Player Example

Figure 9-7 shows a MPEG-4 (.mp4) home video shot from my FlipCam. I simply dragged the MP4 file right onto the surface of the application, and the video began playing. In this section, you learn how to implement a basic JavaFX video player application.



Figure 9-7. A basic JavaFX video player

To create a video player, all you need to do is reuse the previous MP3 audio player example and make very minor adjustments. The code is mostly identical to the MP3 audio player example, except that the visualization (Path) is replaced with a `javafx.scene.media.MediaView` node, as shown in Figure 9-7.

Because the core functionality hasn't changed, you will experience the same features such as seeking, stopping, pausing, and playing media. Another bonus feature added to the video player is the ability to put the video into full-screen mode by double-clicking the application window area. To restore the stage window, repeat the double-click or press the Esc key.

Note Because all video player examples in this chapter were built from the MP3 audio player project, you will notice that features such as seeking, stopping, pausing, and playing media are identical. For the sake of brevity in the remaining examples, I list only the code parts that demonstrate new features and omit sections that existed in the core code from the MP3 Audio Player example.

Video Player Source Code

Listing 9-7 shows the implementation of a JavaFX video player application. After you have examined the code, I will discuss how it works.

Listing 9-7. Source Code for a JavaFX Video Player Application

```
public class PlayingVideo extends Application {

    private MediaPlayer mediaPlayer;
    private Point2D anchorPt;
    private Point2D previousLocation;
    private ChangeListener<Duration> progressListener;
    private BooleanProperty playAndPauseToggle = new SimpleBooleanProperty(true);
    private EventHandler<MouseEvent> mouseEventConsumer = event -> event.consume();

    /**
     * @param args he command line arguments
     */
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        // Bug: JDK-8087498 [Mac] Full screen mode fails for certain StageStyles
        // Remove native window borders and title bar
        primaryStage.initStyle(StageStyle.TRANSPARENT);

        // Create the application surface or background
        Pane root = new AnchorPane();
        root.setId("app-surface");

        Scene scene = new Scene(root, 551, 270, Color.rgb(0, 0, 0, 0));

        // load JavaFX CSS style
        scene.getStylesheets()
            .add(getClass().getResource("/playing-video.css")
                .toExternalForm());
        primaryStage.setScene(scene);

        // Initialize stage to be fullscreen
        initFullScreenMode(primaryStage);

        // ...Code Omitted

        // Create a media view to display video
        MediaView mediaView = createMediaView(root);
    }
}
```

```

        // ...Code Omitted
    }
    /**
     * Bug: JDK-8087498 [Mac] Full screen mode fails for certain StageStyles
     * Attaches event handler code (mouse event) to
     * toggle to Full screen mode. Double click the scene.
     */
    private void initFullScreenMode(Stage primaryStage) {
        Scene scene = primaryStage.getScene();

        // Full screen toggle
        scene.setOnMouseClicked((MouseEvent event) -> {
            if (event.getClickCount() == 2) {
                Platform.runLater(() ->
                    primaryStage.setFullScreen(!primaryStage.isFullScreen()) );
            }
        });
    }

    /**
     * Initialize the stage to allow the mouse cursor to move the application
     * using dragging.
     * @param primaryStage - The applications primary Stage window.
     */
    private void initMovablePlayer(Stage primaryStage) {
        // ...Code Omitted
    }

    /**
     * Initialize the Drag and Drop ability for media files.
     * @param root - The Scene graph's root pane.
     */
    private void initFileDragNDrop(Pane root) {
        // ...Code Omitted
    }

    /**
     * Create a MediaView node.
     *
     * @return Pane
     */
    private MediaView createMediaView(Pane root) {
        MediaView mediaView = new MediaView();
        mediaView.setId("media-view");
        mediaView.setPreserveRatio(true);
        mediaView.setSmooth(true);

        // Calculate width of the MediaView (in AnchorPane)
        DoubleBinding widthProperty = new DoubleBinding(){
            @Override
            protected double computeValue() {

```

```

        return root.getWidth() - (root.getInsets()
            .getLeft() +
            root.getInsets()
            .getRight());
    }
};

// Calculate height of the clipping region.
// This prevents video to display beyond bottom border area.
DoubleBinding heightClipProperty = new DoubleBinding(){
    @Override
    protected double computeValue() {
        return root.getHeight() - (root.getInsets()
            .getTop() +
            root.getInsets()
            .getBottom());
    }
};

// Bindings for the media view's fitWidth
mediaView.fitWidthProperty().bind(widthProperty);

// Recalculate when the width changes
root.widthProperty().addListener(observable1 ->
    widthProperty.invalidate());

// Recalculate when the height changes
root.heightProperty().addListener(observable1 ->
    heightClipProperty.invalidate());

// Create a clip region to be a rounded rectangle
// matching the root pane's rounded border.
// Bindings use the calculated size properties:
// widthProperty and heightClipProperty
Rectangle clipRegion = new Rectangle();
clipRegion.setArcWidth(10);
clipRegion.setArcHeight(10);
clipRegion.widthProperty().bind(widthProperty);
clipRegion.heightProperty().bind(heightClipProperty);
mediaView.setClip(clipRegion);

// Update calculations on invalidation listener
// to reposition media view and clip region
root.insetsProperty().addListener(observable -> {
    widthProperty.invalidate();
    heightClipProperty.invalidate();
    Insets insets = root.getInsets();
    clipRegion.setX(insets.getRight());
    clipRegion.setY(insets.getTop());
    mediaView.setX(insets.getRight());
    mediaView.setY(insets.getTop());
});

```

```
// when errors occur
mediaView.setOnError(mediaErrorEvent -> {
    mediaErrorEvent.getMediaError()
        .printStackTrace();
});

return mediaView;
}

/**
 * Creates a node containing the audio player's
 * stop, pause and play buttons.
 *
 * @param root The Scene graphs root pane.
 * @return Node A button panel having play,
 * pause and stop buttons.
 */
private Node createButtonPanel(Pane root) {
    // ...Code Omitted
}

/**
 * The close button to exit application
 *
 * @return Node representing a close button.
 */
private Node createCloseButton() {
    // ...Code Omitted
}

/**
 * After a file is dragged onto the application a new MediaPlayer
 * instance is created with a media file.
 *
 * @param url - The URL pointing to an audio file.
 * @param root - The scene graph's root pane.
 */
private void playMedia(String url, Pane root) {
    // ...Code Omitted

    // PLEASE ADD THE CODE BELOW
    // Set the media player to display video (MediaView)
    MediaView mediaView
        = (MediaView) root.getScene().lookup("#media-view");
    mediaView.setMediaPlayer(mediaPlayer);

}
```

```

/**
 * Sets play button visible and pause button not visible when
 * playVisible is true otherwise the opposite.
 *
 * @param playVisible - value of true the play becomes visible
 * and pause non visible, otherwise the opposite.
 * @param root - The root node (AnchorPane)
 */
private void updatePlayAndPauseButtons(boolean playVisible, Parent root) {
    // ...Code Omitted
}

/**
 * A position slider to seek backward and forward
 * that is bound to a media player control.
 *
 * @return Slider control bound to media player.
 */
private Slider createSlider() {
    // ...Code Omitted
}
}

```

How It Works

I already mentioned that the `PlayingVideo.java` source code is mostly the same code from the Audio Player project, but with small changes. Let's begin by jumping into those changes. In the `start()` method, the root pane (AnchorPane) is created and styled using the `play-video.css` file. The JavaFX CSS will style the application and controls. To ensure the application is styled correctly be sure to download the source code from the book's web site. After downloading the source code, make sure the resources are in the compiled code, such as `play-video.css`. That way during runtime the resources can be loaded via the classpath.

Setting Up the Stage Window for Full-Screen Mode

```
private void initFullScreenMode(Stage primaryStage)
```

Next, the code invokes the `initFullScreenMode()` method, which attaches handler code to listen for a mouse double-click event. When a user double-clicks the background surface (scene), the application and media view will become full screen. The following code snippet is the handler code set on the `OnMouseClicked` event.

```

private void initFullScreenMode(Stage primaryStage) {
    Scene scene = primaryStage.getScene();
    // Full screen toggle
    scene.setOnMouseClicked((MouseEvent event) -> {
        if (event.getClickCount() == 2) {
            Platform.runLater(() ->
                primaryStage.setFullScreen(!primaryStage.isFullScreen()) );
        }
    });
}

```

The MediaView Node

```
private MediaView createMediaView(Pane root)
```

In the `start()` method after adding the full-screen capability, the code invokes the private method `createMediaView()`, which returns a `MediaView` node that is responsible for displaying the video.

To see what the `createMediaView()` method actually returns, let's look at how it was implemented. Inside the `createMediaView()` method, it begins by instantiating a `MediaView` node with an ID of `media-view`. The code then sets the `preserveRatio` property to true, which preserves the aspect ratio of the video image dimension when it's being resized. In other words, resizing the window won't stretch the video picture. Listing 9-8 creates a new `MediaView` node that preserves the aspect ratio.

Listing 9-8. Creating a `MediaView` Node that Preserves the Aspect Ratio

```
MediaView mediaView = new MediaView();
mediaView.setId("media-view");
mediaView.setPreserveRatio(true);
```

Do you remember when creating the application window by using CSS styling? The reason I ask is that you should notice the background has rounded corners with an inset padding and also has a rounded lined border. Because of these details, I designed it so the video (`MediaView`) would fit inside the rounded border region. As the dimensions of the viewing area could change, such as activating full-screen mode or resizing a window, the code will need to recalculate the width and height values of the view port of the `MediaView` node.

In order to be accurate, I chose to create two custom properties that will dynamically adjust the media view display area. The first custom property is called `widthProperty` and it calculates the media view node's width based on the available space. The available space for the media view's width is calculated based on the root pane's width, minus the left and the right inset values. Insets are the union of the padding and border widths of a pane (root pane).

The second custom property is called `heightClipProperty` and it calculates the value of the `MediaView`'s clip region's height. The height of the video that is being displayed could get larger than the height of the border at the bottom of the application, so a clipping region will be imposed to prevent this from happening. Listing 9-9 shows the two custom properties that compute the display's width and height. If you aren't familiar with custom properties, refer to Chapter 4 on lambdas and properties.

Listing 9-9. Custom Properties to Compute the Width and Height When Displaying the `MediaView` Node

```
// Calculate width of the MediaView (in AnchorPane)
DoubleBinding widthProperty = new DoubleBinding(){
    @Override
    protected double computeValue() {
        return root.getWidth() - (root.getInsets()
            .getLeft() +
            root.getInsets()
            .getRight());
    }
};

// Calculate height of the clipping region.
// This prevents video to display beyond bottom border area.
DoubleBinding heightClipProperty = new DoubleBinding(){}
```

```

@Override
protected double computeValue() {
    return root.getHeight() - (root.getInsets()
        .getTop() +
        root.getInsets()
        .getBottom());
}
};

```

Continuing with the `createMediaView()` method, the code binds the `MediaView` node's `fitWidthProperty` with the custom property `widthProperty`. After binding the `widthProperty`, the code adds invalidation listeners to the root node (`AnchorPane`). These listeners detect a change in the width and height properties of the root node. Once a change is detected, the custom properties `widthProperty` and `heightClipProperty` are marked invalid, which will cause the properties to be recalculated.

Listing 9-10. Bindings and Listeners to Recalculate Custom Properties When the Window Is Resized

```

// Bindings for the media view's fitWidth
mediaView.fitWidthProperty().bind(widthProperty);

// Recalculate when the width changes
root.widthProperty().addListener(observable1 ->
    widthProperty.invalidate());

// Recalculate when the height changes
root.heightProperty().addListener(observable1 ->
    heightClipProperty.invalidate());

```

Next, the code creates a clip region that will create a view port with dimensions that are bound to the two custom properties—`widthProperty` and `heightClipProperty`. Listing 9-11 shows the creation of the clip region that is shaped like the inset rounded border area of the root node's background.

Listing 9-11. The Clip Region Applied to the `MediaView` Node

```

// Create a clip region to be a rounded rectangle
// matching the root pane's rounded border.
// Bindings use the calculated size properties:
// widthProperty and heightClipProperty
Rectangle clipRegion = new Rectangle();
clipRegion.setArcWidth(10);
clipRegion.setArcHeight(10);
clipRegion.widthProperty().bind(widthProperty);
clipRegion.heightProperty().bind(heightClipProperty);
mediaView.setClip(clipRegion);

```

One last listener will be attached to the root pane (`AnchorPane`) to respond whenever the `insets` property changes. Because the application isn't yet shown the inset values are zero. In other words, the CSS styling hasn't been applied and rendered yet at this point. Listing 9-12 shows an invalidation listener to update the clip region and media view properties. Once the CSS is applied and rendered onto the JavaFX scene, the `insets` property will change and call the listener code. If the insets or border widths change, the clip region and media view's position and dimensions will be recalculated.

Listing 9-12. An Invalidation Listener that Responds When the insets Property Changes

```
// Update calculations on invalidation listener
// to reposition media view and clip region
root.insetsProperty().addListener(observable -> {
    widthProperty.invalidate();
    heightClipProperty.invalidate();
    Insets insets = root.getInsets();
    clipRegion.setX(insets.getRight());
    clipRegion.setY(insets.getTop());
    mediaView.setX(insets.getRight());
    mediaView.setY(insets.getTop());
});
```

Lastly, Listing 9-13 shows how the code simply adds handler code to the `OnError` property whenever an error event is raised in the media view node. The code simply prints a stack trace of the media error.

Listing 9-13. Adding Handler Code When the Media View Node Encounters an Error

```
// when errors occur
mediaView.setOnError(mediaErrorEvent -> {
    mediaErrorEvent.getMediaError()
        .printStackTrace();
});

return mediaView;
```

Simulating Closed Captioning: Marking a Position in a Video Media

Have you ever wanted to implement closed captioning video? In the past, there have been discussions about JavaFX 8 possibly supporting the standard closed captioning specification for video media; however, the state of this feature is unclear as of this writing. Fortunately, there is hope; JavaFX supports media event markers to achieve the same behavior as closed captioning video.

Closed Captioning Video Example

To demonstrate a closed captioning video example, I created a project called `ClosedCaptionVideo`. For the sake of space, I will show only the code parts that are relevant to media event markers (see Listing 9-14). To see the full source code, visit the book's web site and download the code for this chapter.

When you read about the video media file formats earlier, you learned that an `.flv` file could be found at the following URL: <http://www.mediacollege.com/adobe/flash/video/tutorial/example-flv.html>.

I use the file `20051210-w50s.flv` for the closed captioning video example. Figure 9-8 depicts the closed captioning (`Label`) being displayed at certain points in the video at the bottom of the screen. The node to display the text is a JavaFX label positioned at the bottom. To change the text font and size, look at the `playing-video.css` file.



Figure 9-8. Closed captioning video

Listing 9-14 shows the code at end of the `playMedia()` method setting up the closed captioning code. Notice that the media object was obtained after the user dragged and dropped a video file over the surface of the application. Next, the code does a lookup to obtain the closed captioning label node to later be updated based on the `OnMarker` event.

Listing 9-14. Media Marker Events

```
// obtain media object
Media media = ...;

// lookup closed caption label
Label closedCaption =
    (Label) root.lookup("#closed-caption-text");

// Rewind back to the beginning
mediaPlayer.setOnEndOfMedia( ()-> {
    updatePlayAndPauseButtons(true, root);
    // change buttons to the play button
    mediaPlayer.stop();
    playAndPauseToggle.set(false);

    // clear closed caption
    closedCaption.setText("");
});

media.getMarkers().put("Starting race",
    Duration.millis(1959));
media.getMarkers().put("He is beginning to get ahead of her.",
    Duration.millis(3395));
media.getMarkers().put("They are turning the corner",
    Duration.millis(6060));
media.getMarkers().put("The crowds are cheering !!!",
    Duration.millis(9064));
media.getMarkers().put("The gentleman makes it to the finish line.",
    Duration.millis(11546));
```

```
// display closed caption
mediaPlayer.setOnMarker((MediaMarkerEvent event) ->
    closedCaption.setText(event.getMarker().getKey())
);
```

How It Works

Listing 9-14 shows a newly created Media object based on a valid URL string. The code begins by looking up the closed captioning Label node.

Assuming the start() method has called the private method `createClosedCaptionArea()`, the root node will contain a Label node with an ID `close-caption-text`. After obtaining the closed captioning node, the code implements handler code when the `OnEndOfMedia` event is raised. The handler code updates the display of the Play and Pause buttons, but also clears the closed captioning Label node.

Next, the code adds key/value pairs as markers in media. Each marker pair consists of a key of type `String` and a value of type `Duration` (in milliseconds). The duration is the time at which an `OnMarker` event is triggered.

You'll immediately notice that the `setOnMarker()` method is set with a lambda expression with a parameter of type `MediaMarkerEvent`. The `MediaMarkerEvent` closure code is invoked when the `OnMarker` event is raised. The closure code simply sets the `closedCaption` (`javafx.scene.control.Label`) node with the key's value as the closed captioning text.

Summary

In this chapter, you discovered the world of JavaFX media. You began by learning about media events and event-based programming when dealing with the JavaFX Media APIs. Next, you saw how easy it is to create a media file as a URL string to be loaded and played. After learning about the `MediaPlayer` API, you got a chance to look at an MP3 audio player example that demonstrated features such as Seek, Stop, Pause, and Play. You also learned how to implement a cool visualization using the `AudioSpectrumListener` interface.

After you learned about audio media, you learned about video. You learned about currently supported media formats such as MPEG-4 and FLV. Then you saw an example of a basic JavaFX video player application. Here, you discovered that playing video is the same as playing audio, except for the use of the `MediaView` node, which is capable of displaying the video. Finally, you were exposed to a short example of media marker events (`MediaMarkerEvent`) to simulate a closed captioning video.

CHAPTER 10



JavaFX on the Web

JavaFX provides capabilities to interoperate with *HTML5* content. The underlying web page rendering engine in JavaFX is the popular open source C++ API called WebKit. This API is used in Apple's Safari browsers, Amazon's Kindle devices, and was used in Google's Chrome browser prior to version 27 (the WebKit fork is called Blink). HTML5 is the de facto standard markup language for rendering content in web browsers. HTML5 content consists of JavaScript, CSS, Scalable Vector Graphics (SVG), Canvas API, Media, XML, and new HTML element tags. In short, you can create JavaFX applications embedded with web browser-like functionality.

The relationship between JavaFX and HTML5 is an important combination because they complement one another by drawing from each of their individual strengths. For instance, JavaFX's rich client APIs coupled with HTML5's rich web content create a user experience resembling a web application with the characteristics of native desktop software. This new breed of application is called the Rich Internet Application (RIA).

Before delving into the example applications, the chapter discusses the following core Java/JavaFX web-based APIs:

- Java 9 Module `javafx.web`, Package namespace: `javafx.scene.web`
 - `WebEngine`
 - `WebView`
 - `WebEvent`
- Java 9 Module: `jdk.jsobject`, Package namespace: `netscape.javascript`
 - `JSONObject`
- Java 9 Module: `jdk.incubator.httpclient`, Package namespace: `jdk.incubator.http`.

The module `jdk.incubator.httpclient` is currently experimental in Java 9.

According to some reports, the module will be finalized in Java 10 and named `java.httpclient`.

- `HttpClient`
- `HttpRequest`
- `HttpResponse`

In this chapter, you examine example applications that do the following:

- Display HTML5 content into a `WebView` node (an SVG-based Analog Clock)
- Communication between Java and JavaScript (`WeatherWidget`)

JavaFX Web and HTTP2 APIs

Before learning about JavaFX Web and HTTP2 APIs, I want to mention that the code examples in this chapter were developed as *Java 9 modules* (Jigsaw). Having said this, I thought it might be a good idea to show you a module definition with the three modules discussed throughout this chapter:

- javafx.web - WebEngine and WebView
- jdk.jsobject - JSObject
- jdk.incubator.httpclient - HttpClient, HttpRequest, HttpResponse

Shown here is a typical application module that depends on these three modules:

```
module com.jfxbe.myapplicationmodule {
    requires javafx.web;
    requires jdk.jsobject;
    requires jdk.incubator.httpclient;
    exports com.jfxbe.myapplicationmodule;
}
```

As a quick reference for the impatient reader, Table 10-1 contains the modules and their class descriptions. I also provide a short code snippet on how to use the various APIs relating to JavaFX Web and HTTP2 APIs in the description column. To see a more detailed discussion related to each class, you can skip Table 10-1.

Table 10-1. Description of Classes Contained Within the Java Modules

Module Name	Class Name	Description
javafx.web	javafx.scene.web.WebEngine	A non-visible UI component capable of loading web content. WebEngine webEngine = new WebEngine(url);
	javafx.scene.web.WebView	A JavaFX node backed with a WebEngine instance capable of rendering HTML5 content to be displayed. WebView webView = new WebView(); webView.getEngine().load("www.oracle.com");
	javafx.scene.web.WebEvent	Callbacks for common HTML browser-based web events. These web events are handled by the WebEngine instance. webView.getEngine() .setOnAlert((WebEvent<String> t) -> { showAlertDialog(t.getData()); });

(continued)

Table 10-1. (continued)

Module Name	Class Name	Description
jdk.jsobject	netscape.javascript.JSObject	The JavaScript bridge object allows Java to talk to the JavaScript engine. When returned, you can set Java objects with the <code>setMember()</code> method to allow JavaScript code to invoke Java methods. <pre>JSObject jsobj = (JSObject) webView. getEngine() .executeScript("window");jsobj.set Member("WeatherWidget", this);</pre>
jdk.incubator. httpclient	jdk.incubator.http.Http Client	A new way to make HTTP requests. Additional features include WebSockets, Authenticators, Proxies, and SSL. <pre>String jsonText = HttpClient. newHttpClient() .send(HttpRequest.newBuilder(URI. create(urlQueryString)) .GET().build(), BodyHandler.asString()).body();</pre>
	jdk.incubator.http.Http Request	The new API to make HTTP requests such as GET, POST, UPDATE, DELETE, and many other methods. <pre>import static jdk.incubator.http. HttpRequest.BodyProcessor.fromString; import static jdk.incubator.http. HttpClient.newHttpClient; HttpRequest req = HttpRequest.newBuilder(URI.create("http://acme/create-account")) .POST(fromString("param1=abc,param2=123")) .build(); newHttpClient().sendAsync(req, BodyHandler.discard(null)) .whenCompleteAsync((resp, throwable) -> { System.out.println("Saving complete."); });</pre>
	jdk.incubator.http.Http Response	After an HTTP request has been made, an HTTP response object is returned. Response objects are populated with a status code, HTTP headers, and the payload or content body. Refer to the Javadoc documentation to see the many methods to convert body content to different data formats. <pre>HttpResponse.BodyProcessor</pre>

Web Engine

JavaFX provides a non-GUI component capable of loading HTML5 content, called the `WebEngine` API (`javafx.scene.web.WebEngine`). This API is basically an object instance of the `WebEngine` class to be used to load a file containing HTML5 content. The HTML5 file to be loaded can be located on a local filesystem, a web server, or inside a JAR file. When you load a file using a web engine object, a background thread is used to load web content so that it does not block the JavaFX application thread. In this section, you learn about the following two `WebEngine` methods for loading HTML5 content:

- `load(String URL)`
- `loadContent(String HTML)`

WebEngine's `load()` Method

The `WebEngine` API, like many APIs that use background threads, adheres to an event-based programming model. Being event-based usually means that method invocations are asynchronous (callbacks); for example, a web engine can load web content asynchronously from a remote web server and notify *handler code* when the content is finished loading.

After being notified, the handler code would then be run on the *JavaFX application thread*. Listing 10-1 loads HTML content from a remote web server in a background worker thread.

Listing 10-1. A `WebEngine` with a `ChangeListener` Lambda that Determines Whether the Background Loading Is Finished (Succeeded)

```
WebEngine webEngine = new WebEngine();
webEngine.getLoadWorker()
    .stateProperty()
    .addListener( (obs, oldValue, newValue) -> {
        if (newValue == SUCCEEDED) {
            // finished loading...
            // process content...
            // Use webEngine.getDocument();
        }
    });
webEngine.load("https://java.com");
```

To monitor or determine whether the worker thread has finished, a `javafx.beans.value.ChangeListener` (lambda expression) can be added to the *state property*. If you are unfamiliar with creating a lambda expression, refer to Chapter 4 on Lambda expressions to be able to use functional interfaces such as the `ChangeListener` or `InvalidationListener` objects.

In Listing 10-1, you'll notice a `ChangeListener` as a lambda expression added to the *state property* via the `addListener()` method. The *state property* contains the status or lifecycle of the web engine's background worker thread. The states are based on the `State` enum, which is owned by the `javafx.concurrent.Worker` interface.

After setting up the listener, the code begins the loading process by using the web engine's `load()` method. Also notice that the string passed into the `load()` method is a valid URL string representing the location of the web content file or web server.

When a ChangeListener lambda expression is added to the state property, the parameters are as follows:

- `obs` is of type `ObservableValue<? extends Worker.State>`, which references the state property on the web engine object.
- `oldValue` is of type `Worker.State enum` and contains the previous value before the state property has changed.
- `newValue` is of type `Worker.State enum` and contains the new state property.

The type specifiers for each parameter of the lambda expression (`ChangeListener`) were left off to allow the parameter signature to be more concise. After the state property changes, the `newValue` parameter can be checked for various thread states. In Listing 10-1, the `newValue` variable is used to check the state of the worker thread, which is an example of eager evaluation.

To be more thorough, I listed all the possible worker thread states:

READY, SCHEDULED, RUNNING, SUCCEEDED, CANCELLED, FAILED

As you saw in Listing 10-1, the code determines whether the worker thread is finished loading the web content, by checking the worker thread's state against the `Worker.SUCCEEDED` enum value.

WebEngine's loadContent() Method

One additional way to load HTML5 content is by using the web engine's `loadContent(String htmlText)` method and accepting a string. What's nice about the `loadContent()` method is that HTML content represented as a string can be pre-generated without having to fetch web pages from a remote server. The following code snippet loads a string representing HTML content:

```
webEngine.loadContent("<html><body><b>JavaFX Rocks!</b></body></html>");
```

Now that you know two strategies to load HTML content, you will see how to obtain and manipulate the raw content, along with common types of data formats that may be obtained from the web engine object after the web content is loaded.

HTML DOM Content

The web engine can load the current page's *document object model* (DOM) as XML content following the W3C standards-based APIs for Java. Typically, developers who are familiar with an XML Document Object Model (DOM) can easily interrogate `org.w3c.dom.Document` objects. This section shows you how to obtain a W3C XML Document object and the raw XML as a `String` object.

Obtaining an org.w3c.dom.Document (DOM) Object

After a web engine instance successfully loads HTML content, an XML DOM can be obtained by invoking the `WebEngine` object's `getDocument()` method.

The following code snippet obtains a document (`org.w3c.dom.Document`) instance, assuming the web engine is finished loading the HTML or XML content.

```
webEngine.getLoadWorker()
    .stateProperty()
    .addListener( (obs, oldValue, newValue) -> {
```

```

        if (newValue == SUCCEEDED) {
            org.w3c.dom.Document xmlDom = webEngine.getDocument();
            // Do something here with xmlDom
        }
    });
webEngine.load("http://myserver.com/my_cool_xml:data.xml");

```

Using Raw XML Content as a String

Depending on the situation, sometimes it is necessary to obtain the raw XML data as a string of text. Developers often use libraries such as *JDOM* or *dom4j* to convert easily between the two data format types. Listing 10-2 shows how to convert from an `org.w3c.dom.Document` object to a `String` object without the need of using the previous mentioned XML parsing libraries.

Listing 10-2. Converting an XML DOM into a String

```

TransformerFactory transformerFactory = TransformerFactory.newInstance();
Transformer transformer = transformerFactory.newTransformer();
StringWriter stringWriter = new StringWriter();
transformer.transform(new DOMSource(webEngine.getDocument()),
    new StreamResult(stringWriter));
String xml = stringWriter.getBuffer().toString();
System.out.println(xml);

```

The output of Listing 10-2 is shown here:

```

<?xml version="1.0"?>
<people>
    <person eyes="green" hair="blonde">
        <firstname>Tracey</firstname>
        <lastname>Dea</lastname>
    </person>
    <person eyes="brown" hair="black">
        <firstname>Carl</firstname>
        <lastname>Dea</lastname>
    </person>
</people>

```

The JavaScript Bridge

Early in this chapter, I discussed how the underlying web engine uses WebKit to load and interact with HTML5 content. What does it mean to interact with HTML5 content? JavaFX's `WebEngine` API has a JavaScript bridge that allows Java code to invoke JavaScript functions or script code inside HTML5 content. So, another question would be how then does one obtain the raw HTML5 content as a string?

Some common use cases out in the field are the following: obtaining raw HTML content for web crawling (used in search engines), caching web pages, screen scraping, data mining, and data analysis. To obtain the raw HTML5, you will need to interact with the JavaScript engine in WebKit to access the web

content's DOM using the `WebEngine`'s `executeScript()` method. The following code statement accesses the HTML document (DOM) to obtain the raw content from the `documentElement.outerHTML` (the top-most element):

```
String html = (String) webEngine.executeScript("document.documentElement.outerHTML");
```

Communicating from Java to JavaScript

Another example of using the method `executeScript()` is shown in Listing 10-3, where the code calls into a JavaScript function `sayHello()` to add text into an existing HTML DIV element with an id of `my_message`. The HTML file containing the function `sayHello()` is shown in Listing 10-4.

Listing 10-3. Java Calling a JavaScript Function

```
// From Java Code
webEngine.executeScript("sayHello('Hi there');");
```

The Java code in Listing 10-3 calls the JavaScript function `sayHello()` to update the DIV element with an id of `my_message`. Listing 10-4 shows the HTML file containing the `sayHello()` function and the DIV element. Notice the code will retrieve the DIV element using the DOM's function `document.getElementById()`. To see more about JavaScript and how to manipulate the DOM, visit:

<https://developer.mozilla.org/en-US/docs/Web/API/Document>

Listing 10-4. An HTML File Containing the `sayHello()` Function

```
<script>
    function sayHello( msg ) {
        document.getElementById('my_message').innerHTML = msg;
    }
</script>
.
.
<div id="my_message"></div>
```

Being able to call JavaScript is a powerful facility that allows Java code to use the `WebEngine` object's JavaScript engine to manipulate the DOM. But what about JavaScript code talking back to Java code?

Communicating from JavaScript to Java

Java 8 allows JavaScript code to make calls into Java code. Allowing JavaScript to call Java code means that web developers can now take advantage of many Java APIs. In this section, we briefly discuss how to allow JavaScript code to communicate with Java code.

Enabling JavaScript to talk to Java is easy. The first step is obtaining the `WebEngine` object's `JSObject` object via a call to the method `executeScript("window")`. The `JSObject` is a wrapper object to allow the bridge between the two languages.

After obtaining the `JSObject`, any object can be passed into the JavaScript engine's context simply by invoking the `setMember()` method. To invoke the `setMember()` method, you pass in a string name and the corresponding object. Assuming the member object has public instance methods, the JavaScript code can access them. Listing 10-5 shows how to set up the `JSObject` for the JavaScript engine to talk to Java.

Listing 10-5. Allowing JavaScript Code to Make Upcalls to Java Code

```
// The web view's web engine
webView.getEngine()
    .getLoadWorker()
    .stateProperty()
    .addListener( (obs, oldValue, newValue) -> {
        if (newValue == Worker.State.SUCCEEDED) {
            // Let JavaScript make upcalls to this (Java) class
            JSObject jsobj = (JSObject) webView.getEngine()
                .executeScript("window");
            jsobj.setMember("ABCD", new HelloWorld() );
        } });
    });

} );
```

You can see in Listing 10-5 that the `HelloWorld` class instance is set as a member in the `(JSObject)` object. Listing 10-6 shows the simple `HelloWorld` class with a `sayGoodbye()` method.

Listing 10-6. Java Class with the Public Method `sayGoodbye()`

```
public class HelloWorld () {
    public String sayGoodbye(String name) {
        return "Hasta la vista " + name;
    }
}
```

After you've set a `HelloWorld` instance as a `JSObject` member named `ABCD`, the JavaScript function `sayGoodbye()` shown in Listing 10-7 can now invoke the Java method `ABCD.sayGoodbye()`.

Listing 10-7. JavaScript Code Calling Java Code

```
.
.
<script>
function sayGoodbye(name) {
    var message = ABCD.sayGoodbye(name);
    document.getElementById('my_message').innerHTML = message;
}
</script>
.
.
<div id="my_message"></div>
```

As a warning, be careful exposing methods that can allow client code to gain access and, thus, potentially harm or view private data.

■ Caution Because JavaScript code has the capability to make calls into Java, it is very important to make sure you aren't exposing APIs that could potentially harm systems or view private data.

As mentioned earlier in the chapter, the `WebEngine` object is a non-GUI component capable of loading web content, so you might be wondering if it is possible to make web service calls capable of retrieving serialized data that has different data formats such as the popular JSON format. To answer this question, you will want to know the `WebEngine` API's responsibilities and limitations.

When using JavaFX's `WebEngine` API to request web content such as HTML, JSON, and XML data, you will encounter some limitations; some of these involve the ability to fetch (GET) data from or submit (POST) data to a web server asynchronously. These types of web requests are called RESTful services.

Of course, there are a handful of interesting ways (hacks) to get JSON data using the `WebEngine` object by calling into the HTML DOM to obtain text content. Listing 10-8 is an example of how to obtain the JSON text content inside of its HTML DOM.

Listing 10-8. Obtaining the JSON Text Inside the DOM

```
webEngine.load("http://myserver.org/current-weather?format=json");
.
.
.

// handler code to obtain JSON data.
String json = (String) webEngine.getDocument()
    .getElementsByName("body")
    .item(0)
    .getTextContent();
```

The handler code traverses the HTML's DOM to obtain text content retrieved from the web engine object. Even though it looks easy to implement to get JSON data as a string, I *do not recommend* using this approach because it is rather fragile and nonstandard. From a code-maintenance standpoint, the code could be taken out of context and a potential bug could surface in the future. I'm sure you've heard the phrase "Just because you can do something doesn't mean you should."

You will quickly find that the `WebEngine` API's responsibility is basically for loading HTML5 and interacting with its content (HTML/JavaScript). The `WebEngine` API is not responsible for making web service calls or serializing data with different transport protocols.

So, is there a better way to make web requests (RESTful)? Of course there is. While there are many libraries you could choose from that simplify web service calls, the next section shows you how to do things using Java 9's new experimental module called `jdk.incubator.httpclient`.

Java 9 Module `jdk.incubator.httpclient`

In the prior edition of this book related to web requests, we used legacy APIs such as the `HttpURLConnection` class; however, in this chapter we replace it in favor of the new experimental Java 9 module `HTTP2` (JSR 110). For many years, Java developers relied heavily on third-party libraries to make web requests in a convenient way. In the previous edition of this book, I provided a pure Java solution by using the legacy `HttpURLConnection` API; however, it had limitations. Here is a brief rundown of the limitations of the `HttpURLConnection` API:

- It supported older protocols that are now obsolete such as gopher, ftp, etc. It does not support WebSockets.
- The APIs are very difficult to use. The APIs don't use a fluent interface API pattern or lambdas expressions for preparing requests and handling error conditions.
- Request calls are in a blocking mode only. The APIs don't use Java 8 new `CompletableFuture` APIs.

The experimental Java 9 module `jdk.incubator.httpclient` contains new APIs to allow users to make HTTP web requests that use Java 8's lambdas and concurrency API `CompletableFuture`. The `CompletableFuture` APIs allow you to make an asynchronous call to later complete. Once the call is complete, the handler code can then be triggered to process the `HttpResponse`. Later in this chapter, we will use these new APIs with an example Weather Widget application capable of making RESTful calls returning JSON data.

In the beginning, the Java architects were going to add the new HTTP2 modules to Java 9; however, as deadlines got tighter for project Jigsaw, it was decided to be pushed out to Java 10. The good news about the new HTTP2 module is that the engineers created an incubator project under the module name `jdk.incubator.httpclient` within JDK 9. The incubator area is for experimentation and will eventually end up as official core APIs. To see the HTTP2 JDK enhancement proposal, visit <http://openjdk.java.net/jeps/110>.

Although the new HTTP2 module in Java 9 is experimental, it is fully functional, as far as I know. The official HTTP2 module implementation in Java 10 will likely be named `java.httpclient`. In fact, once the module moves to Java 10 your code that relies on Java 9's HTTP client module (`jdk.incubator.httpclient`) will likely break and need to point to the new module name. As a quick recap, if you remember from Chapter 2 on modularity, your module definition might look like the following:

```
module com.jfxbe.myclientmodule {
    requires jdk.incubator.httpclient;

    exports com.jfxbe.myclientmodule;
}
```

Assuming you have your `JAVA_HOME` and `PATH` environment variables set, you may not need to add the `jdk.incubator.httpclient` module when executing your application module; however, the `--add-modules` switch is available for adding third-party or platform modules to your module path. To be on the safe side, be sure to add (`--add-modules`) the `httpclient` module when running code, as shown here:

```
$ java --add-modules jdk.incubator.httpclient --module-path mods com.jfxbe.myclientmodule
```

If you aren't familiar with Java 9 modules, refer to Chapter 2 about JavaFX and Jigsaw for more details. Assuming you know about the necessary items, let's continue the discussion of the HTTP2 module.

The new HTTP2 module is mainly divided into two parts—the `HttpClient` and the `HttpRequest` APIs. The `HttpClient` APIs are all about the connections and the `HttpRequest` APIs are about web requests. Both use the builder pattern with a static method `HttpClient.newBuilder()` and `HttpRequest.newBuilder()`, respectively. When following the builder pattern, most methods related to properties allow you to specify values in an ad hoc way. Each method returns the initially created builder instance.

After an invocation of `newBuilder()` on `HttpClient` and `HttpRequest` an instance of a `HttpClient.Builder` and `HttpRequest.Builder` is returned. Once you have finished specifying values on property methods, you will invoke the `build()` method to return the actual instance of the object. In the case with an `HttpClient.Builder` instance and a call to the `build()` method, an `HttpClient` instance is returned to the caller.

HttpClient API

The Java 9 HTTP2 module's `HttpClient` API provides numerous connection settings. According to the Javadoc documentation, an `HttpClient.Builder` supports the following connection settings shown in Table 10-2.

Table 10-2. Connection Settings When Using the HttpClient.Builder Class

Connection Property	Description
authenticator(Authenticator a)	Sets an authenticator to use for HTTP authentication.
cookieManager(CookieManager cookieManager)	Sets a cookie manager.
executor(Executor executor)	Sets the ExecutorService to be used for sending and receiving asynchronous requests.
followRedirects(HttpClient.Redirect policy)	Specifies whether requests will automatically follow redirects issued by the server.
pipelining(boolean enable)	Enables pipelining mode for HTTP/1.1 requests sent through this client.
priority(int priority)	Sets the default priority for any HTTP/2 requests sent from this client.
proxy(ProxySelector selector)	Sets a ProxySelector for this client.
sslContext(SSLContext sslContext)	Sets an SSLContext.
sslParameters(SSLParameters sslParameters)	Sets an SSLParameters.
version(HttpClient.Version version)	Requests a specific HTTP protocol version where possible.

A typical web site that uses basic authentication will specify an instance of a `java.net.Authenticator`. Listing 10-9 shows an `HttpClient` created with a basic authentication.

Listing 10-9. Basic Authentication Specified on an HttpClient Instance

```
HttpClient client = HttpClient.newBuilder()
    .authenticator(new Authenticator() {
        @Override
        protected PasswordAuthentication getPasswordAuthentication() {
            return new PasswordAuthentication("username", "pa$$wOrd1".toCharArray());
        }
    })
    .build();
```

HttpRequest API

After building your HTTP client, you need to create an HTTP request using the `HttpRequest.newBuilder()` method to create an `HttpRequest.Builder` instance. This is where you specify properties to build up a web request. Remember how I mentioned the older `HttpURLConnection` APIs limitations? One of the limitations was that the API was difficult to use. To compare and contrast, let's look at the legacy and improved ways to make a synchronous HTTP GET web request. Listing 10-10 shows the legacy way to make an HTTP GET web request.

Listing 10-10. The Legacy Way to Make an HTTP GET Web Request

```
URL url = new URL(urlQueryString);
HttpURLConnection connection = (HttpURLConnection) url.openConnection(); connection.
setDoOutput(true);
connection.setInstanceFollowRedirects(false);
connection.setRequestMethod("GET");
```

```
connection.setRequestProperty("Content-Type", "application/json"); connection.
setRequestProperty("charset", "utf-8");
connection.connect();
InputStream inStream = connection.getInputStream();
output(inStream);
```

As you can see, the legacy code in Listing 10-10 is verbose. Also, it appears that the http client and request APIs are combined. To see the new and improved code using the new http2 module, look at Listing 10-11 (`HttpRequest`).

Listing 10-11. The Improved Way to Make an HTTP GET Web Request

```
HttpRequest request = HttpRequest.newBuilder(URI.create(queryStr))
    .GET()
    .build();

HttpResponse<String> response = HttpClient.newHttpClient().send(request,
    BodyHandler.asString());

System.out.println(response.statusCode());
System.out.println(response.body());
```

The output of Listing 10-11 should look something like the following:

```
200
<html><body><div>hello world</div></body></html>
```

You will notice in Listing 10-11 that the HTTP GET is a method instead of a string specified on the legacy method `setRequestMethod("GET")`. The new `HttpRequest.Builder` API also allows you to specify a string method by using the `method(String, BodyProcessor)` method. The `method()` method is usually used on the less common HTTP methods such as OPTION or HEAD. Since the HTTP protocol is a standard, the HTTP request builder supports the other commonly used methods, such as PUT, POST, and DELETE.

In Listing 10-11 you'll notice the method `send()` being used as an equivalent blocking behavior to Listing 10-10. In the second scenario, the `send()` method is also making a blocking call to a remote web server. Later, you will learn how to make non-blocking calls to the remote web server.

There are many ways to create `HttpRequest` instances by using an `HttpRequest.Builder` object. To begin, you first must call the `HttpRequest.newBuilder()` method. All the available property methods you can specify on an `HttpRequest.Builder` object are listed in Table 10-3. Remember when you are done specifying property methods on your `HttpRequest.Builder` instance, you will call the `build()` method to return an `HttpRequest` object.

Table 10-3. The `HttpRequest.Builder` Request Property Methods When Using the `HttpRequest.Builder` Class

Request Property	Description
<code>body(HTTPRequest.BodyProcessor reqproc)</code>	Sets a request body for this builder.
<code>copy()</code>	Returns an exact duplicate copy of this builder based on current state.
<code>expectContinue(boolean enable)</code>	Requests server to acknowledge request before sending request body.
<code>followRedirects(HttpClient.Redirect policy)</code>	Specifies whether this request will automatically follow redirects issued by the server.
<code>GET()</code>	Builds and returns a GET <code>HttpRequest</code> from this builder.
<code>header(String name, String value)</code>	Adds the given name/value pair to the set of headers for this request.
<code>headers(String... headers)</code>	Adds the given name/value pairs to the set of headers for this request
<code>method(String method)</code>	Builds and returns an <code>HttpRequest</code> from this builder using the given method <code>String</code> .
<code>POST()</code>	Builds and returns a POST <code>HttpRequest</code> from this builder.
<code>proxy(ProxySelector proxy)</code>	Overrides the <code>ProxySelector</code> set on the request's client for this request.
<code>PUT()</code>	Builds and returns a PUT <code>HttpRequest</code> from this builder.
<code>setHeader(String name, String value)</code>	Sets the given name/value pair to the set of headers for this request.
<code>timeout(TimeUnit unit, long timeval)</code>	Sets a timeout for this request.
<code>uri(URI uri)</code>	Sets this <code>HttpRequest</code> 's request URI.
<code>version(HttpClient.Version version)</code>	Overrides the <code>HttpClient.version()</code> setting for this request.

Now that you've seen how easily you can make synchronous (blocking) web requests using `send()`, let's see how to make RESTful requests asynchronously (non-blocking). Next you will see more examples of how to make RESTful requests using Java 9's experimental module `jdk.incubator.httpclient`.

Making RESTful Requests

In the Enterprise Web development world, you will inevitably encounter the concept of RESTful web services. Based on the concept of representational state transfer (REST), RESTful web services are lightweight HTTP requests that typically involve XML or JSON-formatted data. To make RESTful calls, the web request will contain a request method that tells the web server the type of HTTP data flow transaction.

Like database transactions, RESTful conventions follow the concept of CRUD (Create, Read, Update, and Delete), where the create stage is a POST request, read is a GET, update is a PUT, and delete is a DELETE. While the actions GET and DELETE only involve a query string to make a request, POST and PUT often involve a payload of larger amounts of data as part of the request.

The HTTP GET Request

Because RESTful services are beyond the scope of this book, I will detail only the two common RESTful call requests—GET and POST. Listing 10-12 shows the familiar HTTP GET request that developers typically perform when retrieving JSON data. Keep in mind that in Listing 10-12, the method `sendAsync()` is used as an asynchronous call to the remote web server. This involves a background worker thread fetching the data and later being notified when it is complete.

Listing 10-12. A RESTful GET Request to Retrieve Weather Data

```
import static jdk.incubator.http.HttpClient.newHttpClient;
import static jdk.incubator.http.HttpRequest.newBuilder;
import static jdk.incubator.http.HttpResponse.BodyHandler;
.
.
// =====
// The sendAsync() and whenCompleteAsync() method returns a
// CompletableFuture<HttpResponse<String>> instance.
// =====

String queryStr = "http://myplace/current-weather?q=Pasadena%20MD,US&format=json";
HttpClient.newHttpClient()
    .sendAsync(newBuilder(URI.create(queryStr))
        .GET()
        .build(),
        BodyHandler.asString())
    .whenCompleteAsync( (httpResp, throwable) -> {
        if (throwable != null){
            System.err.println(throwable.getMessage());
            return;
        }
        String json  = httpResp.body();
        System.out.println(json);
    });
}
```

In Listing 10-12, the code begins by creating an `HttpClient` instance via the `newHttpClient()` method. The `newHttpClient()` method is statically imported at the top. This makes the code more concise and more readable by not having to type the owning class. You'll also notice other statically imported methods and classes, such as `newBuilder` and `BodyHandler`.

After creating a new `HttpClient` object, the `sendAsync()` method is invoked with a GET request using the `newBuilder()` method. As a convenience, the `newBuilder()` method follows the builder pattern for a web request to be created. The second (last) parameter to be passed into the `sendAsync()` method is a `BodyHandler` type.

Next, you learn about the body handler type and the `HttpResponse<T>` response objects when the web request is completed.

Body Handlers

For typical web requests, the body handler is simply a `String` type, but there are other body types that can be retrieved. When the body handler is passed in as the second parameter of the `send()` or `sendAsync()` method, the body handler type with respect to the `HttpResponse<T>` returned the type `T` will be the same. For instance, the method `whenCompleteAsync()` will receive an HTTP response having a body handler of type `String`.

In Listing 10-12, the method `whenCompleteAsync()` is triggered upon completing the web request. After being triggered, the handler code will receive an `HttpResponse<String>` and a Java `Throwable`. Again, you will notice the handler code invoking the `httpResp.body()` returning a string. Because the data type from the `BodyHandler.asString()` passed in earlier is a string, the method `body()` returns a value of type `String`.

Do you want to know what other types of body handlers are available?

According to the Java 9 Javadoc documentation, here are some other body handler types that can be retrieved:

- `asByteArray():` Returns a `BodyHandler<byte[]>` that returns a `BodyProcessor<byte[]>` obtained from `BodyProcessor.asByteArray()`.
- `asByteArrayConsumer(Consumer):` Returns a `BodyHandler<Void>` that returns a `BodyProcessor<Void>` obtained from `BodyProcessor.asByteArrayConsumer(Consumer)`.
- `asFileDownload(Path,OpenOption...):` Returns a `BodyHandler<Path>` that returns a `BodyProcessor<Path>` where the download directory is specified, but the filename is obtained from the `Content-Disposition` response header.
The `Content-Disposition` header must specify the attachment type and must also contain a filename parameter. If the filename specifies multiple path components, only the final component is used as the filename (with the given directory name). When the `HttpResponse` object is returned, the body has been completely written to the file and `HttpResponse.body()` returns a `Path` object for the file. The returned `Path` is the combination of the supplied directory name and the filename supplied by the server. If the destination directory does not exist or cannot be written to, then the response will fail with an `IOException`.
- `discard(Object):` Returns a response body handler, which discards the response body and uses the given value as a replacement for it. Used during a `POST` request.
- `asString(Charset):` Returns a `BodyHandler<String>` that returns a `BodyProcessor<String>` obtained from `BodyProcessor.asString(Charset)`. If a charset is provided, the body is decoded using it. If charset is null, then the processor tries to determine the character set from the `Content-encoding` header. If that charset is not supported, `UTF_8` is used.

In this example, an asynchronous call `sendAsync()` method takes a valid URI consisting of a query string. Once the URI is created, the `newBuilder()` method invokes the `GET()` method to specify the HTTP method and finally invokes the `build()` method to return a populated object of type `HttpRequest`. Next, you see the common POST request when the client is sending larger payloads, such as a form submission.

HTTP POST Request

The code to make a RESTful POST request looks like the GET request in the previous example, except that the data submitted will not be specified as an appended query string such as “`?param1=1¶m2=555`”. Instead, to send name/value pairs as a String payload, Listing 10-13 shows an example of a fictitious bank using an HTTP RESTful POST service that creates a new customer bank account.

Listing 10-13. A RESTful POST Request

```
HttpRequest request = HttpRequest.newBuilder(
    URI.create("https://acmebank/create-account"))
    .POST(fromString("id=" + newSeqId + ",firstName=John,lastName=Doe"))
    .build();

HttpClient httpClient = newHttpClient();
httpClient.sendAsync(request, BodyHandler.discard(null))
    .whenCompleteAsync( (httpResp, throwable) -> {
        if (throwable != null){
            System.err.println("Error Saving");
            return;
        }
        System.out.println("Saving complete.");
    });
}
```

In Listing 10-13 the code makes an *asynchronous* call to create a new customer bank account. In this scenario, the id was passed in the POST method using the variable newSeqId instead of being returned in the HTTP response. As a demonstration of the server not returning a created customer id or any string response, the BodyHandler.discard(null) method was specified to ignore body content from the server.

Again, regarding the second parameter of the sendAsync() method as it relates to the data *returned* from the server, if your server code returns an id generated after a record has been created, you'll want to replace the second parameter of the sendAsync() method with BodyHandler.asString(). Once the asynchronous call is completed, you can call the httpResp.body() function to return the message sent from the server.

The new HTTP2 module not only can provide request/response behavior, but it can also offer full-duplex or WebSockets behavior. The next section covers WebSockets.

WebSockets

WebSockets is a protocol for a two-way (full-duplex) communication using a TCP connection. Java 9's http2 module supports the creation of client-side WebSockets. Like the previously mentioned APIs, the WebSockets API also provides a WebSocket.Builder class for easy creation of WebSocket instances.

Server-Side Sockets

This section doesn't show you example code for the server side relating to WebSockets. Since there are literally tons of server-side solutions out in the wild, I suggest you search on the web and pick one. I will basically present client-side code using Java 9's http2 module to talk to a WebSocket that essentially is an *echo* server.

For a WebSocket-based server-side solution, here are some solutions that I found on the Internet:

- Netty: <https://netty.io>
- Vert.x: <http://vertx.io>
- Spark: <http://sparkjava.com>

- JEE 7 vendors:
 - Spring WebSocket: <https://docs.spring.io/spring/docs/current/spring-framework-reference/html/websocket.html>
 - Jetty: <http://www.eclipse.org/jetty>

To test my client-side code, I've used *Spark Java* the embedded web server at <http://sparkjava.com>. They have great examples to create REST APIs and server-side WebSockets. Their documentation shows you how to create a simple echo server that returns what the client sends. To see their implementation of an echo server, visit Spark Java's topic on WebSockets at [http://sparkjava.com/documentation - embedded-web-server](http://sparkjava.com/documentation#embedded-web-server).

To obtain the Spark library, here are the Maven and Gradle coordinates.

Maven coordinates for Spark Java:

```
<dependency>
  <groupId>com.sparkjava</groupId>
  <artifactId>spark-core</artifactId>
  <version>2.6.0</version>
</dependency>
```

Gradle coordinate notation for Spark Java:

```
compile "com.sparkjava:spark-core:2.6.0"
```

Client-Side Sockets

On the client side, you will be creating a WebSocket to connect to the server. To create a WebSocket from an `HttpClient` class, you can invoke the static `newWebSocketBuilder()` method. The method takes two parameters—a URI and a `WebSocket.Listener` object. Listing 10-14 creates a WebSocket connection to an echo server with a URL of `ws://localhost:4567/echo`. By default, Spark uses port 4567.

Listing 10-14. Creating a Client WebSocket to Send Text Over TCP/IP. The Server Will Return the Sent Message Back to the Client

```
HttpClient client = HttpClient.newHttpClient();

WebSocket clientWebSocket = client.newWebSocketBuilder(
    URI.create("ws://localhost:4567/echo"),
    new ClientSocketListener())
.buildAsync().join();

// Send the server two messages
clientWebSocket.sendText("Hello World 1" + new Date());
clientWebSocket.sendText("Hello World 2" + new Date());
```

Listing 10-14 creates an `HttpClient` instance first and then calls the `newWebSocketBuilder()` to create a `WebSocket.Builder` instance. After passing in the URI the second parameter is a new `ClientSocketListener` instance. To finish the builder, a call to `buildAsync()` will return a `CompletableFuture<WebSocket>`. After calling `buildAsync()` a call to the `join()` method will cause it to block and wait to return the `WebSocket` instance. After creating the `WebSocket`, the `sendText()` method is invoked to send text to the server.

The `ClientSocketListener` class shown in Listing 10-15 is responsible for handler code when communicating to the server. The only one that is really implemented is the `onText()` method.

Listing 10-15. An Client Socket Listener Class to Respond to the Echo Server

```
class ClientSocketListener implements WebSocket.Listener {  
    @Override  
    public void onOpen(WebSocket webSocket) {  
        System.out.println("onOpen");  
    }  
  
    @Override  
    public CompletionStage<?> onBinary(WebSocket webSocket, ByteBuffer message, WebSocket.  
MessagePart part) {  
        System.out.println("onBinary");  
        return null;  
    }  
  
    @Override  
    public CompletionStage<?> onPing(WebSocket webSocket, ByteBuffer message) {  
        System.out.println("onPing");  
        return null;  
    }  
  
    @Override  
    public CompletionStage<?> onPong(WebSocket webSocket, ByteBuffer message) {  
        System.out.println("onPong");  
        return null;  
    }  
  
    @Override  
    public CompletionStage<?> onClose(WebSocket webSocket, int statusCode, String reason) {  
        System.out.println("onClose");  
        return null;  
    }  
  
    @Override  
    public void onError(WebSocket webSocket, Throwable error) {  
        System.out.println("onError");  
    }  
  
    @Override  
    public CompletionStage<?> onText(WebSocket webSocket,  
                                    CharSequence message,  
                                    WebSocket.MessagePart part) {  
        System.out.println("onText");  
  
        // request for one message  
        webSocket.request(1);  
  
        // Receive message from server.  
        return CompletableFuture.completedFuture(message)  
              .thenAccept(System.out::println);  
    }  
}
```

In Listing 10-15, the methods are basically all the events and callbacks when messages are received from the server. You will notice that the `onText()` method will return an instance of:

```
CompletableFuture<CharSequence>
```

Because the messages are asynchronous, the `thenAccept()` method is triggered when the message is complete. Once complete, the message of type `CharSequence` will be printed to the console.

Well there you have it, a brief code example on WebSockets. I'm sure you are excited about fetching and posting data; however, maintain that enthusiasm as you'll now see how to display data using JavaFX!

Viewing HTML5 Content (WebView)

Now that you've seen various strategies for fetching web-based content, you will finally get a chance to display HTML5 web content using the powerful JavaFX `WebView` node.

JavaFX provides a GUI `WebView` (`javafx.scene.web.WebView`) node that can render HTML5 content onto the Scene graph. A `WebView` node is basically a mini-browser UI component that can respond to web events and allows a developer to interact with the HTML5 content. Because of the close relationship between loading web content and the ability to display web content, the `WebView` node object also contains a `WebEngine` instance.

To whet your appetite in this section, I've created an example that uses the `WebView` API to render HTML5 content. The example is a JavaFX application that displays an SVG-based analog clock. To obtain the full source code, visit the book's web site and download the `DisplayingHtml5Content` project.

Example: An HTML5 Analog Clock

As an example of using HTML5 content to be rendered onto a `WebView` node, let's look at an analog clock that was created using an SVG file, based on the W3C's scalable vector graphics (SVG) markup language. Figure 10-1 depicts an SVG-based analog clock that was rendered in a JavaFX `WebView` node.



Figure 10-1. SVG-based analog clock (WebView node)

You can see that the analog clock face has hour, minute, and second hands. The clock hands will be initialized and positioned with the current time. Of course, as time goes by, the hands will move in a clockwise rotation.

To download the code, go to the book's web site or to <https://github.com/carldea/jfx9be> for instructions on how to compile and run the code. Listing 10-16 shows statements on how to compile the code, copy resources, and run the displaying HTML5 content application on the DOS command prompt (terminal).

Listing 10-16. The Statements to Compile, Copy Resources, and Run the Displaying HTML5 Content Application Example

```
// Compile
$ javac -d mods/com.jfxbe.html5content $(find src/com.jfxbe.html5content -name "*.java")

// Copy resource
$ cp src/com.jfxbe.html5content/com/jfxbe/html5content/clock.svg \
mods/com.jfxbe.html5content/com/jfxbe/html5content

// Run Application
$ java --module-path mods -m \
com.jfxbe.html5content/com.jfxbe.html5content.DisplayingHtml5Content
```

The compile statement will place the compiled classes into the `mods/com.jfxbe.html5content` directory.

Once you have compiled the code, the next statement will copy the resources into the compiled area. Copying the resources will make files such as the `clock.svg` available during runtime. Usually, your favorite IDE will copy resources automatically. The last statement runs the application by using `--module-path` to point to the newly compiled module (`com.jfxbe.html5content`). Also note the `-m` switch. It tells the Java runtime which module to run and which class contains the application `main()` method.

After successfully running the application, Figure 10-1 shows the `DisplayingHtml5Content` application containing an SVG clock rendered in a `WebView` node. Next, let's look at the source code to see how it was implemented (i.e., what makes it tick?).

Analog Clock Source Code

The source code begins with the module definition shown in Listing 10-17. If you aren't familiar with Java 9's module definitions, refer to Chapter 2 on JavaFX and Jigsaw. If you want to ignore modules and treat the code as Java 8 code, just skip the module definition and look at the main source code shown in Listing 10-18.

Listing 10-17. The Contents of the module-info.java Definition for the `com.jfxbe.html5content` Module

```
module com.jfxbe.html5content {
    requires javafx.web;
    requires jdk.jsobject;
    exports com.jfxbe.html5content;
}
```

The module definition begins with a name such as `com.jfxbe.html5content`. This of course is just a convention and doesn't have to be a namespace and dot separated. As of the writing of this book during compilation I received a warning because my module name has a numeral 5 in `html5content`. At this time, there have been e-mails circulating in regards to naming conventions, overall reduction in the amount of typing when specifying what module names to include, and the simplification of running applications on the command line.

The main analog clock source code is shown in Listing 10-18. The analog clock is based on an SVG file loaded into a JavaFX application’s Scene graph. Listing 10-19 shows the SVG file containing the majority of the functional code that displays the clock face and the JavaScript code that animates the hour, minute, and second hands.

Listing 10-18. Analog Clock Source Code (DisplayingHtml5Content.java)

```
package com.jfxbe.html5content;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.web.WebView;
import javafx.stage.Stage;

import java.net.MalformedURLException;
import java.net.URL;

/**
 * An SVG analog clock rendered using the WebView node.
 * @author cdea
 */
public class DisplayingHtml5Content extends Application {

    @Override
    public void start(Stage primaryStage) throws MalformedURLException {
        primaryStage.setTitle("Displaying Html5 Content");
        WebView browser = new WebView();
        Scene scene = new Scene(browser,320,250, Color.rgb(0, 0, 0, .80));
        primaryStage.setScene(scene);

        URL url = getClass().getResource("clock.svg");
        browser.getEngine().load(url.toExternalForm());
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 10-19 shows the `clock.svg` SVG file that was created using the Inkscape tool. SVG specification is similar to HTML markup, where you can embed JavaScript code to interrogate element tags in the DOM.

Listing 10-19. The `clock.svg` File Containing an SVG Representation of an Analog Clock

```
<svg
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:cc="http://creativecommons.org/ns#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:svg="http://www.w3.org/2000/svg"
  xmlns="http://www.w3.org/2000/svg"
```

```

xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:sodipodi="http://sodipodi.sourceforge.net/DTD/sodipodi-0.dtd"
xmlns:inkscape="http://www.inkscape.org/namespaces/inkscape"
width="300"
height="250"
id="svg4171"
version="1.1"
inkscape:version="0.48.1 "
sodipodi:docname="clock.svg" onload="updateTime()">
<script>
<![CDATA[
var xmlns="http://www.w3.org/2000/svg"
function updateTime()
{
    var date = new Date();
    var hr = parseInt(date.getHours());
    if (hr > 12) {
        hr = hr - 12;
    }
    var min = parseInt(date.getMinutes());
    var sec = parseInt(date.getSeconds());
    var pi=180;
    var secondAngle = sec * 6 + pi;
    var minuteAngle = ( min + sec / 60 ) * 6 + pi;
    var hourAngle = (hr + min / 60 + sec /3600) * 30 + pi;
    moveHands(secondAngle, minuteAngle, hourAngle);
}
function moveHands(secondAngle, minuteAngle, hourAngle) {
    var secondHand = document.getElementById("secondHand"); var minuteHand = document.
getElementById("minuteHand"); var hourHand = document.getElementById("hourHand");
    secondHand.setAttribute("transform", "rotate("+ secondAngle + ")");
    minuteHand.setAttribute("transform", "rotate("+ minuteAngle + ")");
    hourHand.setAttribute("transform", "rotate("+ hourAngle + ")");
} ]]>
</script>
<defs id="defs4173">
... // beginning of SVG code
... // Main clock code
<g id="hands" transform="translate(108,100)"> <g id="minuteHand"> <line stroke-
width="3.59497285" y2="50" stroke-linecap="round" stroke="#00ffff" opacity=".9" />
<animateTransform attributeName="transform" type="rotate" repeatCount="indefinite"
dur="60min" by="360" /> </g>
<g id="hourHand"> <line stroke-width="5" y2="30" stroke-linecap="round" stroke="#ffcb00"
opacity=".9" /> <animateTransform attributeName="transform" type="rotate"
repeatCount="indefinite" dur="12h"
by="360" /> </g> <g id="secondHand">
    <line stroke-width="2" y1="-20" y2="70" stroke-linecap="round" stroke="red"/>
    <animateTransform attributeName="transform" type="rotate" repeatCount="indefinite" dur="60s"
by="360" />
</g> </g>
... // The rest of the Clock code: shiny glare, black button cover (center)
</svg>

```

How It Works

HTML5 allows the use of scalable vector graphics (SVG) content to be shown in browsers. SVG is similar to JavaFX's Scene graph (retained mode), in which nodes can be scaled at different sizes while preserving shape details without the effects of pixelation. In this example, the code in Listing 10-18 represents the Java source code, and Listing 10-19 represents the SVG content code that is loaded into a WebView node.

Before running the example code, make sure the `clock.svg` file is in the build path. In NetBeans, you may need to perform a clean and build before running the application, which will copy the resource `clock.svg` to the build path.

Let's jump right into the code from Listing 10-18, beginning with the `start()` method. This method begins by setting the Stage window's title bar. Then the code instantiates a `WebView` node as the root when creating the `Scene` object. After creation, the scene is set as the Stage window's current scene.

Before loading the file, the code obtains a valid URL string of the location of the file based on the method `getClass().getResources()`.

When loading a URL, you will notice the call to `getEngine().load()`, where the `getEngine()` method will return an instance of a `javafx.scene.web.WebEngine` object. This basically means that all instantiated `WebView` classes will implicitly create their own `javafx.scene.web.WebEngine` instances. The following code snippet is an excerpt from Listing 10-18, showing the JavaFX's `WebEngine` object loading the `clock.svg` file:

```
WebView browser = new WebView();
URL url = getClass().getResource("clock.svg");
browser.getEngine().load(url.toExternalForm());
```

You are probably wondering why the Java source code is so small. It's small because the `WebView` (`javafx.scene.web.WebView`) node is only responsible for rendering HTML5 content. The majority of the code is shown in Listing 10-19, and it contains the `clock.svg` file. This file not only contains SVG markup language, but also the JavaScript code. The SVG specification support animation is applied to elements. Next, let's look at how the SVG content was created.

Inkscape and SVG

The analog clock was created using the popular open source SVG drawing program Inkscape. Because Inkscape is beyond the scope of this book, I will not be discussing the details of the tool. To learn more about Inkscape, visit <http://www.inkscape.org/> for tutorials and demos. To learn more about SVG, visit https://www.w3schools.com/graphics/svg_intro.asp.

In brief, Inkscape allows designers to create shapes, text, and effects to generate illustrations. Because SVG files are considered HTML5 content, these files can also be displayed in an HTML5-capable browser such as the `WebView` node.

Running the example project called `DisplayingHtml5Content`, you have seen that the clock's hands move or animate by rotating clockwise. You are probably wondering how the clock's hands are rotated or positioned. Simply put, when manipulating SVG elements or any HTML5 elements, the JavaScript language will be employed. In fact, in this example the clock's hour, minute, and second hand node elements are rotated by using JavaScript code that exists in Listing 10-19 between the `<script>` tag elements.

Between these `<script>` tags are the JavaScript code statements responsible for initializing and updating the clock's hands. The main code uses the `updateTime()` function to calculate the position of each second, minute, and hour hand. Before the start of the animation of the clock hands, the code sets the clock's initial position by calling the JavaScript `updateTime()` function via the `onload` attribute on the entire

SVG document (located on the root `svg` element). Once the clock's arms are set, the SVG code begins to animate the clock arms by using the `animateTransform` element. The following is SVG code that animates the second hand indefinitely:

```
<g id="secondHand">
    <line stroke-width="2" y1="-20" y2="70"
        stroke-linecap="round" stroke="red"/>
    <animateTransform attributeName="transform"
        type="rotate" repeatCount="indefinite" dur="60s" by="360" />
</g>
```

On a final note, if you want to create a clock like the one depicted in this example, visit <http://screencasters.heathenx.org/blog> to learn about all things Inkscape. Another impressive and beautiful display of custom controls that focuses on clocks, gauges, and dials is the Medusa, Steel Series, and the Enzo library by Gerrit Grunwald. To be totally amazed, visit his blog at <http://harmoniccode.blogspot.com>.

WebEvents

If you've read Chapter 9's discussion of JavaFX Media and how the APIs are designed to be event-driven, you'll be happy to know that the JavaFX Web APIs also adhere to an event-driven programming model. Web events mirror the way browsers raise client-side events such as alerts, errors, and statuses. In this section, you briefly learn about how to respond to web-based events.

If you are familiar with JavaScript client-side web development, you most likely know how to pop up an alert dialog box with a message. If you don't know how to display an alert box, the following JavaScript code snippet will pop up an alert dialog box (assuming you are using a browser):

```
<script>
    alert('JavaFX is Awesome');
</script>
```

Typically, old-school web developers will use alert boxes to debug web pages. However, modern web developers more likely use the JavaScript function `console.log()` to output information into a console provided by Firebug or the Chrome browser's developer tools. So, what do you think would happen if an HTML web page that contained the `alert()` code just shown was loaded into a `WebView` node?

When the code is executed inside a JavaFX `WebView` node, a native dialog window will not pop up. But the `OnAlert` event does get raised as a `javafx.scene.web.WebEvent` object. To set up the handler, use the `setOnAlert()` method with an inbound parameter of type `WebEvent`.

The following code snippet will receive `WebEvent<String>` objects containing the string message to be output onto the console:

```
webView.getEngine().setOnAlert((WebEvent<String> wEvent) -> {
    System.out.println("Alert Event - Message: " + wEvent.getData());
    // ... show a JavaFX Alert dialog window
});
```

Developers who want to create their own dialog windows upon an alert-triggered message can simply respond to `WebEvents` and use JavaFX Dialog APIs such as an `Alert` instance. Table 10-4 shows all the `WebEngine`'s web events.

Table 10-4. javafx.scene.web.WebEngine WebEvents and Properties

Set On Method	On Method Property	Description
setOnAlert()	onAlertProperty()	JavaScript alert handler
setOnError()	onErrorProperty()	WebEngine error handler
setOnResized()	onResizedProperty()	JavaScript resize handler
setOnStatusChanged()	onStatusChanged()	JavaScript status handler
setOnVisibilityChanged()	onVisibilityChangedProperty()	JavaScript window visibility handler
setConfirmHandler()	confirmHandlerProperty()	JavaScript Confirm window

Now that you know how to retrieve data, manipulate HTML, and respond to browser-based web events, let's look at an example that combines all the techniques that you've learned so far. This example is a Weather Widget application that fetches JSON data from a RESTful endpoint. You will also see bidirectional communication between Java and JavaScript using the JavaScript bridge object `netscape.javascript.JSObject`.

Weather Widget Example

The example demonstrated in this section is a Weather Widget that combines all the web-based concepts mentioned so far. The example is a JavaFX application that loads JSON weather information from the Open Weather Map API at <http://openweathermap.org>. Figure 10-2 shows the Weather Widget example with the current weather conditions in Columbus Ohio, USA.

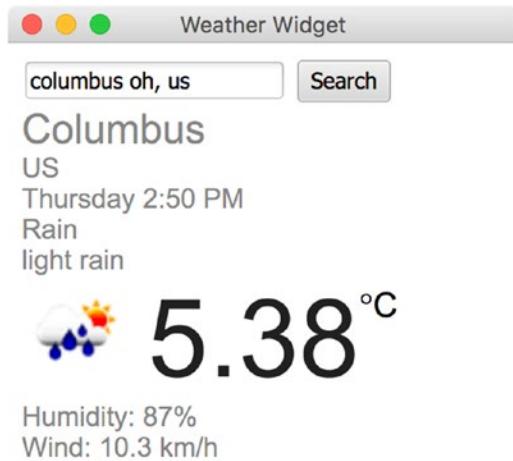


Figure 10-2. A Weather Widget displaying the current weather condition. Once you've entered the city, state code, and country code in the format of "CITY STATE, COUNTRY," you click the Search button to fetch the JSON data.

To download the code, go to the book's web site or to <https://github.com/carldea/jfx9be> for instructions on how to compile and run the code. Because the http2 module is part of the incubator area, I want to bring it to your attention when you include the module when compiling and running a warning. Listing 10-20 shows the statements for how to compile the code, copy resources and run the Weather Widget application on the DOS command prompt (terminal).

Listing 10-20. The Statements to Compile, Copy Resources, and Run the Weather Widget Application Example

```
// Mac OSX and Linux
// Compile
$ javac -d mods/com.jfxbe.weatherwidget $(find src/com.jfxbe.weatherwidget -name "*.java")

// Copy resource
$ cp src/com.jfxbe.weatherwidget/com/jfxbe/weatherwidget/weather_template.html \
mods/com.jfxbe.weatherwidget/com/jfxbe/weatherwidget

// Run Application
$ java --add-modules jdk.incubator.httpclient --module-path mods -m com.jfxbe.weatherwidget/
com.jfxbe.weatherwidget.WeatherWidget

// Windows OS
// Compile
c:\jfxbe\chap10> javac -d mods\com.jfxbe.weatherwidget src\com.jfxbe.weatherwidget\module-
info.java src\com.jfxbe.weatherwidget\com.jfxbe.weatherwidget\WeatherWidget.java

// Copy resource
c:\jfxbe\chap10> copy src\com.jfxbe.weatherwidget\com\jfxbe\weatherwidget\weather_template.html \
mods\com.jfxbe.weatherwidget\com\jfxbe\weatherwidget

// Run Application
c:\jfxbe\chap10> java --add-modules jdk.incubator.httpclient --module-path mods -m com.
jfxbe.weatherwidget/com.jfxbe.weatherwidget.WeatherWidget
```

The compile statement will place the compiled classes into the `mods/com.jfxbe.weatherwidget` directory. When compiling, you should see the following warning on the console:

```
warning: using incubating module(s): jdk.incubator.httpclient
1 warning
```

You can ignore the warning and proceed with the other steps. The warning basically indicates that it is considered experimental and not part of the core.

Once you have compiled the code, you need to copy resources into the compiled area. Copying the resources will make files available during runtime. Usually your favorite IDE will copy resources automatically. The last statement runs the application by adding the `jdk.incubator.httpclient` module via `--add-modules` and using the `--module-path` to point the newly compiled module to the `mods` directory. Note the `-m` switch, which tells the Java runtime which module to run and which class contains the `main` application. In this case, the module name and main class name are specified like so:

```
com.jfxbe.weatherwidget/com.jfxbe.weatherwidget.WeatherWidget.
```

After successfully running the application, Figure 10-2 shows the Weather Widget application window.

Before I get too far ahead relating to the source code, I want to point out that for the Weather Widget application to successfully retrieve data, a valid API key must be obtained from the folks at Open Weather Map. To register and get an API key, go to <http://openweathermap.org>. If this is the first time you've run the application, you will be prompted with a dialog window asking you to enter in the API key, as shown in Figure 10-3.

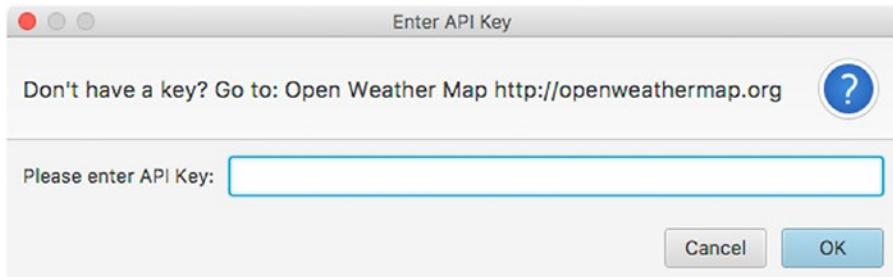


Figure 10-3. A dialog to enter the API key from the Open Weather Map

Once the API key is entered and accepted, it will be stored locally in a file named:

```
{user.home}/.openweathermap-api-key
```

Where {user.home} is where you substitute your home directory. Once the file is saved, when you launch the application again, the API key file will be loaded instead of prompting the user for the API key.

In Java's system properties, user.home is the user's home directory depending on the operating system. See the Javadoc documentation and tutorial at the following:

<https://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html>

Note The web service in this example is powered by the people of Open Weather Map (<http://openweathermap.org>). I encourage you to support them as they continue to provide a valuable service with maps and weather data around the world for application and mobile developers.

One-Liner: Reading an Input Stream into a String

As a really nice trick for how the Weather Widget example loads the API key file, I created a simple static utility method `streamToString()`. This handy utility method allows you to read a file's content or a connection's input stream and convert it into a string.

When you look at the Weather Widget example's source code, the method will attempt to read a local file as a string containing an API key issued from the web site <http://openweathermap.org>.

Listing 10-21 shows how to convert an input stream into a String object.

Listing 10-21. A Static Method that Reads a Text File Returned as a String Object to the Caller

```
private static String streamToString(InputStream inputStream) {
    String text = new Scanner(inputStream, "UTF-8")
        .useDelimiter("\\Z")
        .next();
    return text;
}
```

You will quickly notice the strange one-line statement that “*magically*” converts an input stream into a `String` object. The `streamToString()` method uses the convenient `Scanner` API. This API is excellent for reading text files that have delimited text lines, such as comma-separated values (CSV). `Scanner` objects default their delimiter as an end-of-line character (`\n`).

By setting the end-of-file (EOF) marker (`\Z`) as the delimiter, the `next()` method will return the text stream before the end, creating the entire text as one field. Of course, there are other ways to retrieve text data, and probably more efficient ways, too; however, you can see that using this strategy can handle common situations with small- to medium-sized data requests (up to ~3MB). The size can be an issue because the data is converted to a `String` object and, depending on the number of requests without garbage collection, memory resources can be easily consumed—possibly overwhelming the application. Since it is only an API key consisting of a hash, it should be perfectly fine.

The Weather Widget allows users to enter a city or region into the search field to determine the weather condition. After fetching the JSON weather data, the screen will display the following information: City, Country, Day of Week, Time (Last update), Weather Condition, Weather Description, Icon, Temperature, Humidity, and Wind Speed. The code will execute the JavaScript code to populate the display.

Okay, enough with the pre-requisites and description of the application—let’s now look at the Weather Widget’s source code.

Source Code

The Weather Widget project consists of three source files:

- `module-info.java`: Module definition requires `jdk.incubator.httpclient` etc.
- `WeatherWidget.java`: Main application code
- `weather_template.html`: The HTML template used as the main display of weather info

In Listing 10-22, the `module-info.java` file contains the module definition. The Weather Widget module depends on the following modules: `javafx.web`, `jdk.jsobject`, and `jdk.incubator.httpclient`. The last statement is the exports of the Weather Widget module itself. Exporting a module is how to make the module and its APIs public to whoever uses it. For instance, if another module adds a requires `com.jfxbe.weatherwidget`, their code can safely import packages owned by `com.jfxbe.weatherwidget`. Remember that transitive dependencies don’t have to be included in the module definition if an existing module already references it, such as the module `java.base`.

Listing 10-22. The `module-info.java` for the `com.jfxbe.weatherwidget` Module

```
module com.jfxbe.weatherwidget {
    requires javafx.web;
    requires jdk.jsobject;
    requires jdk.incubator.httpclient;

    exports com.jfxbe.weatherwidget;
}
```

Listing 10-23 is the main Weather Widget application code in the `WeatherWidget.java` file. In the source code, I've omitted some imports to save some space in this chapter, but I've included the static imports to help make code later more concise. It's a good idea to read the main comments at the top of the class to better understand the data flow when the code communicates back and forth between Java and JavaScript.

Listing 10-23. Source Code of the Weather Widget (`WeatherWidget.java`)

```
package com.jfxbe.weatherwidget;

// Other imports omitted...

import static jdk.incubator.http.HttpClient.newHttpClient;
import static jdk.incubator.http.HttpRequest.newBuilder;
import static jdk.incubator.http.HttpResponse.BodyHandler;
/***
 * The WeatherWidget application demonstrates the
 * use of the following interactions:
 * <pre>
 * 1) Communications from Java to JavaScript
 * 2) Communications from JavaScript to Java
 * 3) RESTful GET Web service end point
 * 4) Manipulate JSON Objects
 * 5) Handle HTML/JavaScript WebEvents
 * 6) Debugging using Firebug lite
 * </pre>
 *
 * <pre>
 *      The following are the steps to help
 *      demonstrate the above interactions:
 *
 * Step 1: The user enters a city state and country into the
 *         search text field. (See weather_template.html)
 * Step 2: After search button is pressed the JavaScript function
 *         findWeatherByLocation() is called. (See weather_template.html)
 * Step 3: An up call from JavaScript to Java is made to the method
 *         WeatherWidget.queryWeatherByLocationAndUnit(). (See this class)
 * Step 4: After querying the weather data the JSON data is passed to
 *         the Java method populateWeatherData(). This method will call
 *         the JavaScript function populateWeatherData(). (See this class)
 * Step 5: Populates the HTML page with the JavaScript function
 *         populateWeatherData() (See weather_template.html)
 *
 * </pre>
 *
 * It is required to obtain a valid API key to query weather data.
 * To obtain an API key head over to Open Weather Map at
 * http://openweathermap.org
 *
 * @author cdea
 */
public class WeatherWidget extends Application {
```

```
/** The main URL of the current weather REST end point. */
public static final String WEATHER_URL = "http://api.openweathermap.org/data/2.5/weather";

/** A local file containing a valid API key. */
public static final String API_KEY_FILE = ".openweathermap-api-key";

/** The API key to access weather and map data. */
private static String API_KEY = null;

/** The Weather display HTML page */
public static final String WEATHER_DISPLAY_TEMPLATE_FILE = "weather_template.html";

/** A WebView node to display HTML5 content */
private WebView webView;

/** A singleton http client to make http requests */
private static HttpClient HTTP_CLIENT;

@Override
public void start(Stage stage) {
    stage.setTitle("Weather Widget");

    webView = new WebView();

    Scene scene = new Scene(webView, 300, 300);
    stage.setScene(scene);

    // obtain API key
    loadAPIKey();

    // Turns on Firebug lite for debugging
    // html,css, javascript
    // enableFirebug(webView);

    // The web view's web engine
    webView.getEngine()
        .getLoadWorker()
        .stateProperty()
        .addListener( (obs, oldValue, newValue) -> {
            if (newValue == Worker.State.SUCCEEDED) {
                // Let JavaScript make up calls to this (Java) class
                JSObject jsobj = (JSObject)
                    webView.getEngine()
                        .executeScript("window");
                jsobj.setMember("WeatherWidget", this);
                // default city's weather (a sunny place)
                queryWeatherByLocationAndUnit("Miami,FL", "c");
            }
        });
}
```

```

// Display a JavaFX dialog window explaining the error
webView.getEngine().setOnAlert((WebEvent<String> t) -> {
    System.out.println("Alert Event - Message: " + t.getData());
    showErrorDialog(t.getData());
});

// Load HTML template to display weather
webView.getEngine()
    .load(getClass()
        .getResource(WEATHER_DISPLAY_TEMPLATE_FILE)
        .toExternalForm());
stage.show();
}

/**
 * If an API key file doesn't exist prompt the user to enter their key.
 * Once a valid key is saved into a file named .openweathermap-api-key
 * The application will use it to fetch weather data.
 */
private void loadAPIKey() {
    // Load API key from local file
    File keyFile = new File(System.getProperty("user.home") + "/" + API_KEY_FILE);

    // Check for file's existence and read/write privileges.
    if (keyFile.exists() && keyFile.canRead() && keyFile.canWrite()) {
        try (FileInputStream fis = new FileInputStream(keyFile)){
            Optional<String> apiKey = Optional.ofNullable(streamToString(fis));
            apiKey.ifPresent(apiKeyStr -> API_KEY = apiKeyStr);
        } catch (Exception e) {
            Platform.exit();
            return;
        }
    } else {
        // If the API key does not exist display dialog box.
        TextInputDialog dialog = new TextInputDialog("");
        dialog.setTitle("Enter API Key");
        dialog.setHeaderText("Don't have a key? Go to: Open Weather Map http://openweathermap.org");
        dialog.setContentText("Please enter API Key:");

        Optional<String> result = dialog.showAndWait();

        result.ifPresent(key -> {
            // write to disk
            try (FileOutputStream fos = new FileOutputStream(keyFile)) {
                String apiKey = result.get();
                fos.write(apiKey.getBytes());
                fos.flush();
                API_KEY = apiKey;
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
    }
}

```

```

        Platform.exit();
        return;
    }
});
}
}

/**
 * Display error dialog window.
 * @param errorMessage
 */
private void showErrorDialog(String errorMessage) {
    Alert dialog = new Alert(Alert.AlertType.ERROR, errorMessage);
    dialog.setTitle("Error Retrieving Weather Data");
    dialog.show();
}

@Override
public void stop() throws Exception {
    // clean up resources here...
}

/**
 * Quick one liner that delimits on the end of file character and
 * returning the whole input stream as a String.
 * @param inputStream byte input stream.
 * @return String A string from an input stream.
 */
public String streamToString(InputStream inputStream) {
    String text = new Scanner(inputStream, "UTF-8")
        .useDelimiter("\\Z")
        .next();
    return text;
}

/**
 * Returns a string containing the URL with parameters.
 * @param cityRegion The city state and country. State and
 *                   country is separated by a comma.
 * @param unitType Specify c for celsius and f for fahrenheit.
 * @return String A query string representing the web request.
 */
private String generateQueryString(String cityRegion, String unitType) {
    String units = "f".equalsIgnoreCase(unitType) ? "imperial": "metric";

    String queryString = WEATHER_URL +
        "?q=" + cityRegion +
        "&" + "units=" + units +
        "&" + "mode=json" +
        "&" + "appid=" + API_KEY;
    return queryString;
}
}

```

```

/**
 * This method is called from the JavaScript function
 * findWeatherByLocation(). Refer to the weather_template.html
 * file.
 * <pre>
 * -- Step 3 --
 * </pre>
 *
 * @param cityRegion The city, state and country.
 * @param unitType The temperature in celsius or fahrenheit.
 */
public void queryWeatherByLocationAndUnit(String cityRegion,
                                           String unitType) {

    // build a weather request
    String queryStr = generateQueryString(cityRegion, unitType);
    System.out.println("Request (http2): " + queryStr);

    // Make a GET request to fetch weather data asynchronously
    // The sendAsync() method returns a CompletableFuture<HttpServletResponse<String>>
    HTTP_CLIENT.sendAsync( newBuilder(URI.create(queryStr))
        .GET()
        .build(),
        BodyHandler.asString())
        .whenCompleteAsync( (httpResp, throwable) -> {
            if (throwable != null){
                showErrorDialog(throwable.getMessage());
                return;
            }
            String json = httpResp.body();
            populateWeatherData(json, unitType);
            System.out.println("Response (http2): " + json);
        });
}

/**
 * Invokes to the JavaScript function populateWeatherData() using the web engine.
 * <pre>
 * -- Step 4 --
 * From Java a call to invoke a JavaScript function is made by
 * calling populateWeatherData().
 * </pre>
 *
 * @param json The JSON string to be evaluated (converted to a real JavaScript object).
 * @param unitType The symbol and unit for the temperature.
 */
private void populateWeatherData(String json, String unitType) {
    Platform.runLater(() -> {
        // On the JavaFX Application Thread....
        webView.getEngine()
            .executeScript("populateWeatherData(eval(" + json + "), " +
                           "'"+ unitType + "' );");
    });
}

```

```

public static void main(String[] args){
    HTTP_CLIENT = newHttpClient();
    Application.launch(args);
}
}
}

```

Lastly, Listing 10-24 consists of the HTML and JavaScript code in a file resource `weather_template.html`. This file is loaded into the JavaFX WebView node.

Listing 10-24. An HTML Template File Populated with Weather Data

```

<!DOCTYPE html>
<html>
<head>
    <title>Weather Widget</title>
    <style type="text/css">
        body { background-color:#ffffff; }

        .tileTextDisplay {
            font-family: arial,sans-serif;
            font-weight: lighter !important;
            color: #878787 !important;
        }
        .largerFont {
            font-size: x-large !important;
        }
        .mediumFont {
            font-size: medium !important;
        }
        #weather-temp {
            font-family: 'Arial', sans-serif;
            font-size: 64px;
            color: #212121 !important;
        }
        #unitType {
            font-family: 'Arial', sans-serif;
            font-size: 20px;
            color: #000000 !important;
        }
    </style>

    <script>
var weekday=new Array(7);
weekday[0]="Sunday";
weekday[1]="Monday";
weekday[2]="Tuesday";
weekday[3]="Wednesday";
weekday[4]="Thursday";
weekday[5]="Friday";
weekday[6]="Saturday";

```

```

/* -- Step 2 --
 * Up call to Java invoking the
 * WeatherWidget.queryWeatherByLocationAndUnit() method.
 */
function findWeatherByLocation() {
    var cityInfo = encodeURIComponent(document.getElementById('search-field').value);
    setInnerText("error-msg", "");
    WeatherWidget.queryWeatherByLocationAndUnit(cityInfo, "c");
}

/* -- Step 5 --
 * Populate UI with weather data.
 * This function is called from Java code.
 */
function populateWeatherData(json, unitType) {
    var jsonWeather = json;
    if (jsonWeather.cod) {
        if (jsonWeather.cod != 200) {
            document.getElementById('error-msg').innerHTML = jsonWeather.message;
            alert(jsonWeather.message);
            return;
        }
    }

    setInnerText('city', jsonWeather.name);
    setInnerText('country', jsonWeather.sys.country);

    var weatherTime = new Date(jsonWeather.dt * 1000);
    var timeStr = timeFormat(jsonWeather.dt * 1000);

    setInnerText('weather-day-time', weekday[weatherTime.getDay()] + " " + timeStr) + "
(Last Update)";
    setInnerText('weather-current', jsonWeather.weather[0].main);
    setInnerText('weather-current-desc', jsonWeather.weather[0].description);
    document.getElementById('weather-icon').src = "http://openweathermap.org/img/w/" +
    jsonWeather.weather[0].icon + ".png";
    setInnerText('weather-temp', jsonWeather.main.temp);
    setInnerText('weather-humidity', "Humidity: " + jsonWeather.main.humidity + "%");
    var windSpeed = (unitType === 'f') ? 'mph' : 'km/h';
    setInnerText('weather-wind-speed', "Wind: " + jsonWeather.wind.speed + " "
+windSpeed);
}

function setInnerText(id, text) {
    document.getElementById(id).innerText = text;
}

function timeFormat( millis ) {
    var weatherTime = new Date(millis);
    var hours = weatherTime.getHours();
    var minutes = weatherTime.getMinutes();
    var meridian = hours >= 12 ? 'PM' : 'AM';
}

```

```

hours = hours % 12;
hours = hours ? hours : 12; // hour '0' means '12'
minutes = minutes < 10 ? '0' + minutes : minutes;
var timeStr = hours + ':' + minutes + ' ' + meridian;
return timeStr;
}

function init() {
    // Apply listener to override enter key press.
    document.getElementById('search-field')
        .addEventListener("keypress", function(event){
            // Overide default enter key to fetch and
            // populate weather info.
            if(event.keyCode==13) {
                event.preventDefault();
                findWeatherByLocation();
                return;
            }
        });
}
</script>
</head>
<body id="weather_background" onload="init();">
<form>
    <input id="search-field" placeholder="City State, Country Code" type="text"
        name="searchField"/>
    <!-- Step 1 - The user enters a city state and country -->
    <input id="search-button" type="button" onclick="findWeatherByLocation()"
        name="searchButton" value="Search"/>
</form>
<div id="error-msg"></div>
<div id="city" class="tileTextDisplay largerFont"></div>
<div id="country" class="tileTextDisplay mediumFont"></div>
<div id="weather-day-time" class="tileTextDisplay mediumFont"></div>
<div id="weather-current" class="tileTextDisplay mediumFont"></div>
<div id="weather-current-desc" class="tileTextDisplay mediumFont"></div>
<div>
    <img id="weather-icon" style="float:left;height:64px;width:64px" src="" alt=""/>
    <div id="weather-temp" style="padding-left:10px; float:left;"></div>
    <div style="float:left; font-size:20px; margin-top:6px">
        <span style="display:inline">&deg;<span id="unitType">C</span></span>
    </div>
</div>
<div style="clear:both;"></div>
<div id="weather-humidity" class="tileTextDisplay mediumFont"></div>
<div id="weather-wind-speed" class="tileTextDisplay mediumFont"></div>
</body>
</html>

```

How It Works

The code is divided into two parts—the Java code shown in Listing 10-23 and HTML/JavaScript code shown in Listing 10-24. The purpose of this example is to demonstrate the bidirectional communication between Java and JavaScript while providing RESTful HTTP GET requests of JSON data. Because all of its aspects have already been discussed, from retrieving data to manipulating HTML, I will basically discuss the high-level steps of the `start()` method of the JavaFX WeatherWidget code from Listing 10-23.

The following code steps were taken within the `start()` method from the source code in Listing 20-23.

1. Set up the scene and stage window with a `WebView` as the root.
2. Read the API key file. If the API key doesn't exist, show the prompt dialog window.
3. Create a `WebView` node to display HTML content.
 - a. Create handler code that responds when the `WebView`'s content is finished loading. The page `weather_template.html` is only loaded once as a single page application.
 - b. Once loaded, the code passes the application instance (`this`) into the web engine to allow up calls from JavaScript code. To allow JavaScript code to access Java objects, the code invokes the `setMember()` method by passing in the name and object, as shown:

```
// Let JavaScript make upcalls to this (Java) class
JSObject jsobj = (JSObject) webView.getEngine().executeScript("window");
jsobj.setMember("WeatherWidget", this);
```

- c. Default to the weather forecast of Miami, Florida, by fetching JSON weather data to be populated into the HTML page. The lowercase "c" is for the unit type, degrees Celsius. By calling the following Java method, it will fetch JSON data and call into JavaScript code to populate the HTML DOM with the weather data.

```
queryWeatherByLocationAndUnit("Miami%20FL,US", "c");
```

4. Create handler code for the `OnAlert` property on the `WebView`'s web engine. The handler code responds to HTML's JavaScript `alert()` function for `WebEvent`. If there are errors returned from the web server, the web event data will be passed to my private `showAlertDialog()` method. This method will display a dialog window with the error message.

```
webView.getEngine().setOnAlert( (WebEvent<String> t) -> {
    System.out.println("Alert Event - Message: " + t.getData());
    showErrorDialog(t.getData());
});
```

5. Load the HTML file `weather_template.html` from the classpath to be displayed onto the `WebView`.

```
// Load HTML template to display weather
webView.getEngine()
    .load(getClass().getResource(WEBER_DISPLY_TEMPLATE_FILE)
        .toExternalForm());
```

In Listing 10-23, you'll notice the comment section regarding Steps 1 through 5; these show the application flow starting from the user searching for weather based on a city, state, and country, to displaying the data in the HTML page. The following are the steps from 1 to 5:

Step 1: The button click invokes the JavaScript function
`findWeatherByLocation()`:

```
<input id="search-button" type="button" onclick="findWeatherByLocation()" name="searchButton" value="Search"/>
```

Step 2: The upcall to the actual Java code method
`queryWeatherByLocationAndUnit()`:

```
function findWeatherByLocation() {...}
```

Step 3: The Java method that uses the new `http2` module to make an asynchronous RESTful GET to fetch weather data:

```
public void queryWeatherByLocationAndUnit(String cityRegion, String unitType)
```

Step 4: After the JSON data is fetched (on the Java side), a call to the `executeScript()` method passes the JSON object into the JavaScript function `populateWeatherData()`:

```
private void populateWeatherData(String json, String unitType) {
    Platform.runLater(() -> {
        // On the JavaFX Application Thread....
        webView.getEngine()
            .executeScript("populateWeatherData(eval(" + json + "), " +
                "'" + unitType + "'');");
    });
}
```

Step 5: The HTML page is populated with the JSON data representing the weather:

```
function populateWeatherData(json, unitType) {...}
```

Enhancements

As an exercise, you may want the Weather Widget application to fetch new weather data periodically. According to the Open Weather Map, their weather forecast updates are typically every three hours.

As a hint on how might you might call the `setInterval()` function, check out this code snippet:

```
// Every period query and populate the forecast using cached URL
setInterval(function(){
    WeatherWidget.queryWeatherByLocationAndUnit(null, "c");
},
(1000 * 60 * 60 * 3) // every 3 hours
); // set interval
```

Summary

In this chapter, you learned about the `WebEngine` API for retrieving web HTML5 content. You began by learning two ways to load web content—using the `WebEngine`'s `load()` and `loadContent()` methods. After learning about how to load web content, you saw how to obtain raw content such as a DOM XML or XML. Next, you learned about the powerful JavaScript bridge that allows bidirectional communication between Java and JavaScript. Getting familiar with web development concepts, you also learned about some of the limitations to JavaFX's `WebEngine` API ability to make web requests.

Within these limitations on making RESTful web service calls, you got a chance to learn about Java 9's new experimental `httpclient` module. By using the `httpclient` module, you can make RESTful GET and POST requests with builder classes, thus simplifying the creation of web requests.

You also learned about the popular WebSockets protocol. Here, you got a chance to create client-side web sockets to talk to an echo server. After learning the many ways to retrieve data and manipulate HTML, you saw an example on how to render HTML5 content by running an animated SVG-based analog clock application.

Finally, you saw all the concepts combined into the last example of a Weather Widget application using JavaFX's `WebView` node. The Weather Widget makes calls to retrieve JSON data from openweathermap.org to then be displayed on an HTML5 page in a `WebView` node. This example's core focus was to learn how Java code can communicate with the `WebEngine`'s JavaScript bridge.

Now that you have a taste of JavaFX and the web, let's move on to some advanced topics. In the next chapter, you explore the world of JavaFX 3D!

CHAPTER 11



JavaFX 3D

JavaFX 9 strongly supports a true 3D scene with adjustable lights, camera, and models as first class citizens of the language. With JavaFX 9, it is possible to mix these capabilities directly with the more well known 2D GUI components mentioned in this book, both visually and in the source code. The advent of dedicated GPU hardware has lowered the barrier to specific language support for 3D rendering such that the performance in JavaFX is on par with third-party implementations of previous versions of Java.

A small book could be dedicated to all the capabilities of the 3D graphics tools within JavaFX 9. For the sake of introductory or intermediate learner, this chapter focuses on quickly establishing scenes with true three-dimensional visualizations of simple objects. These concepts are then expanded upon by describing typical customizations that would be found in more complex scenes.

In this chapter, you learn about the following:

- Geometry primitives
- Cameras and lights
- Custom TriangleMesh models
- Event handling and interacting with 3D scenes

Basic 3D Scenes in JavaFX

For this half of the 3D chapter, you will use a growing example model. It takes you through several steps, from establishing a basic empty 3D scene to a scene with objects that can be viewed from different distances and angles.

A Very Basic 3D Scene Example

The best part about the new JavaFX 3D support is that it is a first class member of the Java language. Care was taken in designing the 3D packages so that very little changes in paradigm from their 2D counterparts. The difference in establishing a basic 3D scene is merely in the camera. Listing 11-1 demonstrates this.

Listing 11-1. Initializing a 3D Scene with Camera

```
/* From SimpleScene3D.java */

Group sceneRoot = new Group();
Scene scene = new Scene(sceneRoot, sceneWidth, sceneHeight);
scene.setFill(Color.WHITE);
PerspectiveCamera camera = new PerspectiveCamera(true);
camera.setNearClip(0.1);
camera.setFarClip(10000.0);
camera.setTranslateZ(-1000);
scene.setCamera(camera);

primaryStage.setTitle("SimpleScene3D");
primaryStage.setScene(scene);
primaryStage.show();
```

Adding this code snippet to an otherwise empty JavaFX application will give you a full 3D scene. Run the application and you should see a view such as Figure 11-1.



Figure 11-1. A basic empty JavaFX 3D scene

Being an empty scene, it is not very exciting; however, you will be adding some objects soon. The example shows that any 3D scene is only differentiated from a 2D layout by two lines:

```
PerspectiveCamera camera = new PerspectiveCamera(true);
scene.setCamera(camera);
```

Creating a 3D camera node such as a `PerspectiveCamera` creates a separate scene graph node, much in the same way you might create a `BorderPane` or `Button`. In that sense, it must be added to the scene like any other node. However, the camera node is special in that it is added via the `scene.setCamera()` method. The Boolean parameter for the `PerspectiveCamera` constructor determines where the 3D origin is. Setting this to true will establish all 3D layout from the origin of (0,0,0) so that all axes cross in the center of the screen. This is typical for most 3D scenes you would expect to build using JavaFX. Setting this parameter to false establishes the 3D origin at the upper-left corner, similar to a 2D component layout. For most scenes, `PerspectiveCamera(true)` is ideal. Some scenes, especially scientific data displays, are more natural with an upper-left origin.

Note A good rule of thumb for selecting the camera type is, if you plan on translating and rotating the camera around the scene much, you should use `PerspectiveCamera(true)`.

Primitives

Like most 3D graphics libraries, JavaFX 9 comes with several out-of-the-box geometric shapes that can be added to a scene. These shapes are commonly referred to as *primitives* and can be added to a scene in the same way that any traditional node type could be added.

Adding a Primitive Example

This section extends the previous very simple scene example so that you can see something. To do this, you need to create a `Cylinder` object, give it a colored material, and add it to the scene. Listing 11-2 shows the code that demonstrates this.

Listing 11-2. Adding a Cylinder Primitive to a JavaFX 3D Scene

```
/* SimpleScene3D.java */

//Step 1b: Add a primitive
final Cylinder cylinder = new Cylinder(50, 100);

final PhongMaterial blueMaterial = new PhongMaterial();
blueMaterial.setDiffuseColor(Color.DEEPSKYBLUE);
blueMaterial.setSpecularColor(Color.BLUE);
cylinder.setMaterial(blueMaterial);

sceneRoot.getChildren().add(cylinder);

//End Step 1b
```

Adding this code to the previous example will add the cylinder and colorize it. Recompile and run the application and you should see a view like Figure 11-2.

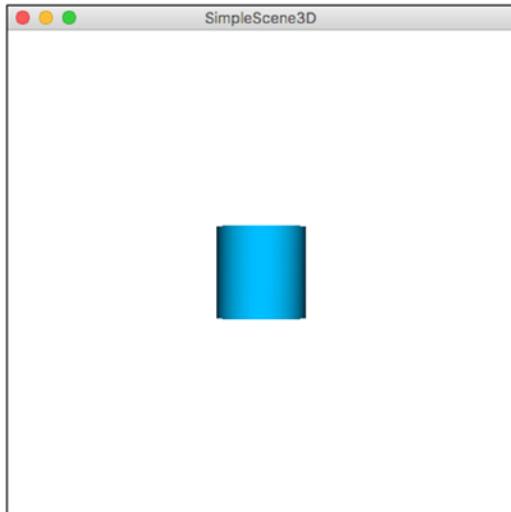


Figure 11-2. Cylinder with PhongMaterial and DrawMode.FILL

This is better than an empty screen. The `Cylinder` constructor used here:

```
final Cylinder cylinder = new Cylinder(50, 100);
```

specifies the diameter and height of the shape. To achieve that deep color blue with that fancy specular highlight, it uses:

```
final PhongMaterial blueMaterial = new PhongMaterial();
```

`PhongMaterial` is an extension of the `javafx.paint` hierarchy and allows for a rich smooth interpolation over 3D surfaces. A `PhongMaterial` object can be reused between primitives and other 3D nodes, as described later. If you do not want to use a `PhongMaterial`, the default rendering for a primitive or any `Shape3D` object is a wireframe view. This will be demonstrated in later examples. Once your shape is constructed and colored, you can add it to your scene with:

```
sceneRoot.getChildren().add(cylinder);
```

In this line of code, the `Cylinder` primitive is treated like any other graphic `Node` you might use or create. The `Cylinder` primitive is added to the scene by default at the origin, or $(0,0,0)$.

Simple Translate and Rotate Example

Now that you have something to look at, you can utilize the third dimension by transforming it. JavaFX 3D supports all the standard 3D transformations you would expect with easy-to-use classes. The following transformations, subclassed from the `Transform` class, all support 3D nodes and groups:

- Affine
- Rotate
- Scale
- Shear
- Translate

Typically in any 3D scene you would be most interested in `Rotate` and `Translate`, which are most concerned with moving and changing the orientation of objects, lights, and the camera. JavaFX `Shape3D` objects such as the `Cylinder` class provide setter methods to easily apply a simple rotation or translation. Listing 11-3 demonstrate how apply this to the `Cylinder` in this scene.

Listing 11-3. Translate and Rotate Primitive into Position

```
/* SimpleScene3D.java */

//Step 1c: Translate and Rotate primitive into position
cylinder.setRotationAxis(Rotate.X_AXIS);
cylinder.setRotate(45);
cylinder.setTranslateZ(-200);
//End Step 1c
```

Adding the code snippet from Listing 11-3 to the current 3D scene application will generate a view where the cylinder has been rotated and translated toward the camera. This should be similar to Figure 11-3.

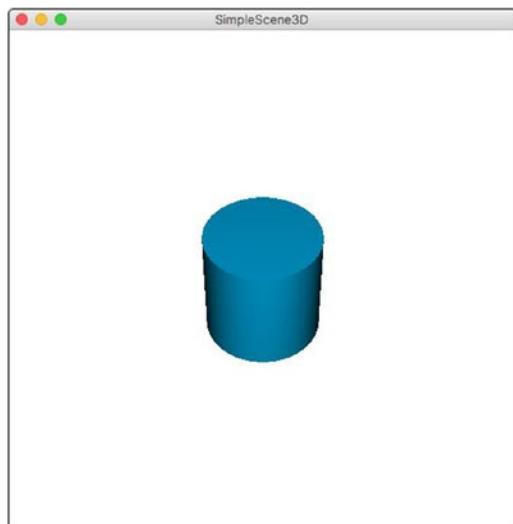


Figure 11-3. A cylinder rotated 45 degrees on the x-axis

For the first time in this running example, you can see the third dimension using a Rotation on the X-axis and a Translation in the Z-axis. Rotations are set in degrees with a range of 0 to 360 degrees. It is important to first set the current rotation axis using `setRotationAxis()`. If you need multiple rotations in multiple axes, you need to call the `setRotationAxis()` method each time the axis changes. Each call to `setRotationAxis()` affects only the current `Shape3D` object. Translations are set in pixels along the given axis of the set command and are handled in the same way that traditional 2D components are translated.

Note The coordinate system for a JavaFX 3D scene uses a “Y Pointing Down” which is the Y-axis is positive down. As with most 3D systems, the Z-axis is positive toward the screen and the X-axis is positive to the right.

Multiple Primitive Transformation Example

This example expands on the current setup by adding and transforming the Cube and Sphere primitives in the same scene as Cylinder. The example code is provided in Listing 11-4.

Listing 11-4. Adding and Transforming Box and Sphere Primitives

```
/* SimpleScene3D.java */

//Step 1d: Add and Transform more primitives
final PhongMaterial greenMaterial = new PhongMaterial();
greenMaterial.setDiffuseColor(Color.DARKGREEN);
greenMaterial.setSpecularColor(Color.GREEN);
final Box cube = new Box(50, 50, 50);
cube.setMaterial(greenMaterial);
final PhongMaterial redMaterial = new PhongMaterial();
redMaterial.setDiffuseColor(Color.DARKRED);
redMaterial.setSpecularColor(Color.RED);
final Sphere sphere = new Sphere(50);
sphere.setMaterial(redMaterial);

cube.setRotationAxis(Rotate.Y_AXIS);
cube.setRotate(45);
cube.setTranslateX(-150);
cube.setTranslateY(-150);
cube.setTranslateZ(150);

sphere.setTranslateX(150);
sphere.setTranslateY(150);
sphere.setTranslateZ(-150);
sceneRoot.getChildren().addAll(cylinder, cube, sphere);
//End Step 1d
```

This code snippet shows multiple materials and colors being applied to multiple primitives. Each primitive is translated and rotated to different depths and axes of the 3D scene.

Having multiple Shape3D objects in the same scene demonstrates a great visual depth. However, as you added multiple nodes or groups to your 3D scene, you must change your `add()` to an `addAll()`, as in the following line from Listing 11-4:

```
sceneRoot.getChildren().addAll(cylinder,cube,sphere);
```

In that sense there is no difference between adding 2D GUI components as scene children and here in 3D. Adding the prior code snippet to the example and making the change to `addAll()` should provide a view similar to Figure 11-4.

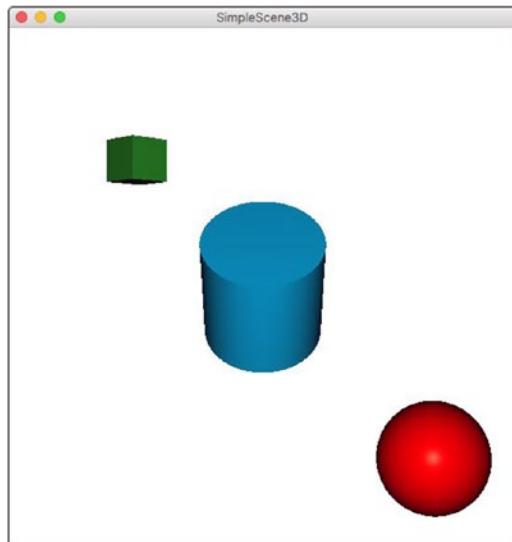


Figure 11-4. A Cube, Cylinder, and Sphere primitive

All Together Now: Grouped Primitives

As your scenes become more complicated, it will become cumbersome to add each 3D object to your scene. Primitives and other Shape3D nodes are often used in conjunction to form more complex objects. These composite objects are typically transformed as one, although each component is often visualized independently. This is simply done using a group, as demonstrated in Listing 11-5.

Listing 11-5. Grouping and Transforming Primitives

```
/* SimpleScene3D.java */

//Step 1e: All Together Now: Grouped Primitives
Group primitiveGroup = new Group(cylinder,cube,sphere);
primitiveGroup.setRotationAxis(Rotate.Z_AXIS);
primitiveGroup.setRotate(180); //Rotate the Group as a whole
sceneRoot.getChildren().addAll(primitiveGroup);
//End Step 1e
```

Each primitive is added to the Group object using the following line:

```
Group primitiveGroup = new Group(cylinder,cube,sphere)
```

Many bulk operations can then be simplified. Listing 11-5 demonstrates a rotation about the Z-axis of 180 degrees, which effectively rotates the entire scene as one homogenous unit. Adding the previous code snippet to the running primitives example should provide a view like Figure 11-5.

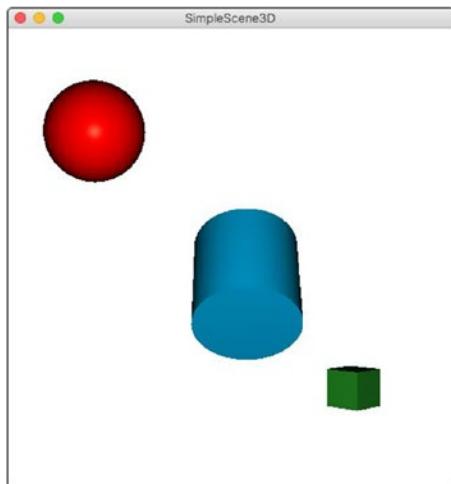


Figure 11-5. Grouped primitives rotated uniformly

Be sure to add the new Group object to your scene instead of your individual primitives, as noted in Listing 11-5. Individual primitive objects can still be accessed, rotated, and translated independently via the Group object's `getChildren()` method or by direct reference.

Note Remember when adding primitives, groups, and other `Shape3D` node objects to a 3D scene, the default is the origin. Most primitives are centered on the origin by a calculated center of mass.

Interacting with Your Scene

Rendering objects into 3D scenes is typically only useful if you can interact with that scene. At a minimum the user should have the ability to move through the scene and do a little sight seeing. This section of the chapter will walk through an example of a simple first person perspective set of controls. The primitive example from the previous section is used as a platform to build from.

Primitive Picking for Primitives

When interacting directly with the individual objects of a 3D scene, typically with a mouse, you need to detect mouse type events and whether those events are related to something within your scene. The good news is event handling within a 3D scene is established in the same pattern as a traditional 2D GUI. Listing 11-6 demonstrates the code to add a `MouseClicked` event handler.

Listing 11-6. Translate a Primitive by Mouse Event

```
/* SimpleScene3D.java */

//Step 2a: Primitive Picking for Primitives
scene.setOnMouseClicked(event-> {
    Node picked = event.getPickResult().getIntersectedNode();
    if(null != picked) {
        double scalar = 2;
        if(picked.getScaleX() > 1)
            scalar = 1;
        picked.setScaleX(scalar);
        picked.setScaleY(scalar);
        picked.setScaleZ(scalar);
    }
});
//End Step 2a
```

Adding this event handler to the scene in the running example creates an interaction where clicking a given 3D object will either double its size or reduce it back to normal. Clicking once on each primitive should provide a view similar to Figure 11-6.

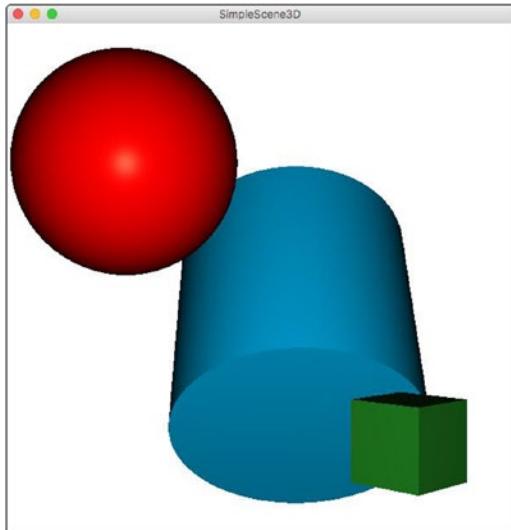


Figure 11-6. 3D objects scaled up in all three axes

The trick here is in determining whether the MouseEvent occurred in congruence with a 3D object in the scene. The magic line of code is:

```
Node picked = event.getPickResult().getIntersectedNode();
```

where the MouseEvent returns a PickResult. A PickResult in 3D speak is a test of intersection of X,Y screen coordinates onto X,Y,Z 3D Scene coordinates. While PickResults were available in JavaFX 2 for 2D GUI components, in JavaFX 9 they are functional not only in the 3D space but as simple accessible objects from various events. As with all MouseEvent types, the event can be queried for a PickResult. If a PickResult occurred then the getIntersectedNode() will return the Node object representation of the scene object “picked”. If no object was “picked” then getIntersectedNode() returns null.

Note Do not forget to scale objects equally in all three axes, as shown in Listing 11-6. Otherwise, your object will quickly become stretched in odd ways.

First Person Movement Using the Keyboard

A convenient way to navigate through a 3D scene is through a First-Person perspective where you “fly through”. Often this is done through a combination of mouse and keyboard where the keyboard navigates the X- and Z-axes and the mouse directs the camera to rotate. This is a typical arrangement for games of this nature but also simulations and tactical displays.

To add keyboard navigation through a 3D scene you must simply add the standard JavaFX key event handlers. The code in Listing 11-7 is a simple example of an OnKeyPressed event handler that provides the X- and Z-axes navigation described previously. The movement produced by this handler translates the Camera node itself through the 3D scene.

Listing 11-7. A Simple First Person Movement Keyboard Event Handler

```
/* SimpleScene3D.java */

//Step 2b: Add a Movement Keyboard Handler
scene.setOnKeyPressed(event -> {
    double change = cameraQuantity;

    //Add shift modifier to simulate "Running Speed"
    if(event.isShiftDown()) {
        change = cameraModifier;
    }

    //What key did the user press?
    KeyCode keycode = event.getCode();

    //Step 2c: Add Zoom controls
    if(keycode == KeyCode.W) {
        camera.setTranslateZ(camera.getTranslateZ() + change);
    }
}
```

```

if(keycode == KeyCode.S) {
    camera.setTranslateZ(camera.getTranslateZ() - change);
}

//Step 2d: Add Strafe controls
if(keycode == KeyCode.A) {
    camera.setTranslateX(camera.getTranslateX() - change);
}
if(keycode == KeyCode.D) {
    camera.setTranslateX(camera.getTranslateX() + change);
}
});
//End Step 2b-d

```

Breaking down the code snippet, you can see JavaFX makes this very simple and straightforward. Adding the event handler to your scene is the same as adding an event handler to any traditional 2D GUI component demonstrated with `scene.setOnKeyPressed(event -> {`.

Adding composite event handling, i.e. holding the Shift key while pressing another key, is simple via the event object itself. In Listing 11-7 this is done by `event.isShiftDown()`, which returns a simple Boolean as to the truth. These additional meta keys are either true or false regardless of what actual event occurred and can be utilized whether the event is keyboard, mouse, or even touch.

When determining which way to move the camera, simply acquire the `KeyCode` from the event:

```
KeyCode keycode = event.getCode();
```

and branch your logic on a simple equality check:

```

if(keycode == KeyCode.W) {
    camera.setTranslateZ(camera.getTranslateZ() + change);
}

```

`KeyCode.W` is part of the `KeyCode` enum that provides every possible key with a `Comparable` interface. Very convenient for building complex interfaces to any application.

First Person Camera Movement Using the Mouse

The previous keyboard handler example provided half of a first-person perspective where you “fly through” your 3D scene. The keyboard is used to slide along the X- and Z-axes. What is happening is the `PerspectiveCamera` node is translating along those axes. However, the previous example limits these translations to a single plane. To simulate looking around, the “head on a swivel” effect, you need to use the mouse.

To add mouse control of the camera swivel for a 3D scene, you must simply add a JavaFX `OnMouseMoved` event handler. The code in Listing 11-8 is a rudimentary example of an `OnMouseMoved` event handler and some basic trigonometry that swivels the camera node. The movement produced by this handler rotates the `Camera` node itself on the Y- and X-axes.

Listing 11-8. A Simple First-Person Camera Rotate Control MouseEvent Handler

```

/* SimpleScene3D.java */

//Step 3: Add a Camera Control Mouse Event handler
scene.setOnMouseMoved(event -> {
    //acquire the new Mouse coordinates from the recent event
    double mouseYnew = event.getSceneY();
    double mouseXnew = event.getSceneX();
    //Find the deltas for our Trig formula
    double atan2Y = mouseYnew - mouseYold;
    double atan2X = mouseXnew - mouseXold;
    //Build the new Rotate objects
    Rotate xRotate = new Rotate();
    Rotate yRotate = new Rotate();
    //Only compute when there is a change
    if(mouseYnew != mouseYold) {
        //When looking up or down we should rotate on the X-AXIS
        camera.setRotationAxis(Rotate.X_AXIS);
        //calculate the rotational change of the camera pitch
        double pitchRotate = camera.getRotate() + (Math.atan2(atan2Y,atan2X) /
        rotateModifier);
        //set min/max camera pitch to prevent camera flipping
        pitchRotate = pitchRotate > cameraYlimit ? cameraYlimit : pitchRotate;
        pitchRotate = pitchRotate < -cameraYlimit ? -cameraYlimit : pitchRotate;
        //replace the old camera pitch rotation with the new one.
        //camera.setRotate(pitchRotate); //This doesn't work when a second rotation
        //call is made
        xRotate = new Rotate(pitchRotate,Rotate.X_AXIS);
    }
    if(mouseXnew != mouseXold) {
        //When looking left or right we should rotate on the Y-AXIS
        camera.setRotationAxis(Rotate.Y_AXIS);
        //calculate the rotational change of the camera yaw
        double yawRotate = camera.getRotate() + (Math.atan2(atan2X,atan2Y) /
        rotateModifier);
        //camera.setRotate(yawRotate); //this would replace the previous X-AXIS
        //rotation
        yRotate = new Rotate(yawRotate, Rotate.Y_AXIS);
    }
    //Apply the combined rotations
    camera.getTransforms().addAll(xRotate,yRotate);
    mouseXold = mouseXnew;
    mouseYold = mouseYnew;
});
//End Step 3

```

This code snippet is by far the single largest addition made so far to the running example; however, most of it is needed to facilitate a smooth swivel feel for the camera node. You will see that some trigonometric functions are used to calculate rotations and this can get tricky quick.

These rotations are applied in an alternative manner than previously in this running example. Before covering the details of how to rotate the camera, let's examine the mouse event handling itself:

```
scene.setOnMouseMoved(event -> {
```

JavaFX 3D makes adding a `MouseEvent` handler to the scene very simple and straightforward. Like all input events, we can acquire the X and Y coordinates from the `MouseEvent`. The goal here is to convert the change of mouse position to a rotation for the camera. You can accomplish this with a little trigonometry:

```
double atan2Y = mouseYnew - mouseYold;
double pitchRotate = camera.getRotate() + (Math.atan2(atan2Y,atan2X) / rotateModifier);
```

This is a very rudimentary method for doing this. Most high-performance first-person games and interactive environments use more exotic calculations and include the many corner cases that tend to occur from use. One of these corner cases is vertical rotation beyond 180 degrees. Left alone, the camera would spin without limit vertically sometimes referred to as “somersaults” or “camera whip.” Found in Listing 11-8 are the following lines of code that will effectively limit camera rotation on the X-axis, which visually appears to be the Y-axis:

```
pitchRotate = pitchRotate > cameraYlimit ? cameraYlimit : pitchRotate;
pitchRotate = pitchRotate < -cameraYlimit ? -cameraYlimit : pitchRotate;
```

Note Remember dealing with what a camera “sees” is opposite from how the camera is positioned within the scene. For example “looking up” along the Y-axis is actually a rotation on the X-axis.

Finally, when dealing with multiple repeated transformations, especially rotations, a better way to apply these transformation to an object is using the object’s `getTransforms().addAll()` method like so:

```
camera.getTransforms().addAll(xRotate,yRotate);
```

This will guarantee that all your rotate, translate, and scale transformations are applied as a composite transformation. If you were to call `setRotate` or `setTranslate` multiple times for each rotation, each subsequent `setRotate` call would replace the previous one. The end result would be a rotation or translation that is equal only to the last transformation called in sequence.

Beyond the Basics

This half of the 3D chapter shows you how to grow a new example focused on ways to provide custom renderings and interactions. Some of the concepts discussed in the first half will be built upon. This section will explain `TriangleMesh` objects, how to “wind” a 3D object using triangles, and to add a `TriangleMesh` object to a 3D scene. You will then learn how to convert this to reusable code, which can allow for many customizations that could be the start of your own library of 3D primitives. Finally, the section explores an alternative camera arrangement that is suitable for examining centralized objects.

Custom 3D Objects Using the TriangleMesh Class

Sometimes the prepackaged 3D primitives provided by JavaFX 9 aren't enough to build the scene you want. It is true that many real-world objects could be represented by a composite of these primitives using the Group type and some clever transformation. However, using only the base primitives would be cartoonish and very cumbersome to maintain within a codebase. Most practical applications require custom shapes and models and rendering must be fast. The good news is JavaFX 9 provides a means to do this using the `TriangleMesh` class. The `TriangleMesh` class is a `Shape3D` class just like `Box`, `Cylinder`, and `Sphere`, which provides direct access to the points (vertices), faces, and texture coordinates collections that make up the rendered geometry. Under the hood of the base three primitives is a `TriangleMesh` where all points and faces are computed based on relevant parameters. To start this custom `TriangleMesh` example, you will need a similar starting application like the previous example, as shown in Listing 11-9.

Listing 11-9. Initializing a 3D Scene with Camera

```
/* From TriangleMeshes.java */
public class TriangleMeshes extends Application {

    private PerspectiveCamera camera;
    private final double sceneWidth = 600;
    private final double sceneHeight = 600;
    private double scenex, sceney = 0;
    private double fixedXAngle, fixedYAngle = 0;
    private final DoubleProperty angleX = new SimpleDoubleProperty(0);
    private final DoubleProperty angleY = new SimpleDoubleProperty(0);

    .....//Step 1: Build your Scene and Camera
    Group sceneRoot = new Group();
    Scene scene = new Scene(sceneRoot, sceneWidth, sceneHeight);
    scene.setFill(Color.BLACK);
    PerspectiveCamera camera = new PerspectiveCamera(true);
    camera.setNearClip(0.1);
    camera.setFarClip(10000.0);
    camera.setTranslateZ(-1000);
    scene.setCamera(camera);

    primaryStage.setTitle("TriangleMeshes");
    primaryStage.setScene(scene);
    primaryStage.show();
    //End Step 1
}
```

“Winding” and Wuthering

For this custom object, we will build and render something that is geometrically simple like a Square Pyramid. A square pyramid has five faces: four equilateral vertical triangle faces and a square base. This will allow you to focus on the `TriangleMesh` interface rather than the complex math. The goal of a `TriangleMesh` is to define process by which a series of triangles are rendered to give the illusion of a continuous 3D surface. Each triangle is connected and defined by a set of vertices in the 3D scene. The method is called “winding” because when defining the vertices of each triangle, the order that the vertices are declared indicates to the camera when and how to render the face of that triangle. To explain the process of constructing and viewing a `TriangleMesh` object, we will break down a method that uses parameters to construct a square pyramid. The complete method is shown in Listing 11-10.

Listing 11-10. A Reusable Method for Building a Custom Pyramid TriangleMesh

```

/* From TriangleMeshes.java */

    //Step 2a: Create a general Pyramid TriangleMesh building method with height and
    hypotenuse
    private Group buildPyramid(float height, float hypotenuse,
        Color color,
        boolean ambient,
        boolean fill) {
    final TriangleMesh mesh = new TriangleMesh();
    //End Step 2a
    //Step 2b: Add 5 points, later we will build our faces from these
    mesh.getPoints().addAll(
        0,0,0,          //Point 0: Top of Pyramid
        0,height,-hypotenuse/2, //Point 1: closest base point to camera
        -hypotenuse/2,height,0, //Point 2: left most base point to camera
        hypotenuse/2,height,0, //Point 3: farthest base point to camera
        0,height,hypotenuse/2 //Point 4: right most base point to camera
    ); //End Step 2b
    //Step 2c:
    //for now we'll just make an empty texCoordinate group
    mesh.getTexCoords().addAll(0,0);
    //End Step 2c
    //Step 2d: Add the faces "winding" the points generally counter clock wise
    mesh.getFaces().addAll( //use dummy texCoords
        0,0,2,0,1,0, // Vertical Faces "wind" counter clockwise
        0,0,1,0,3,0, // Vertical Faces "wind" counter clockwise
        0,0,3,0,4,0, // Vertical Faces "wind" counter clockwise
        0,0,4,0,2,0, // Vertical Faces "wind" counter clockwise
        4,0,1,0,2,0, // Base Triangle 1 "wind" clockwise because camera has rotated
        4,0,3,0,1,0 // Base Triangle 2 "wind" clockwise because camera has rotated
    ); //End Step 2d
    //Step 2e: Create a viewable MeshView to be added to the scene
    //To add a TriangleMesh to a 3D scene you need a MeshView container object
    MeshView meshView = new MeshView(mesh);
    //The MeshView allows you to control how the TriangleMesh is rendered
    meshView.setDrawMode(DrawMode.LINE); //show lines only by default
    meshView.setCullFace(CullFace.BACK); //Removing culling to show back lines
    //End Step 2e
    //Step 2f: Add it to a group, this will be useful later
    Group pyramidGroup = new Group();
    pyramidGroup.getChildren().add(meshView);
    //End Step 2f
    //Step 2g: Customizing your Pyramid
    if(null != color) {
        PhongMaterial material = new PhongMaterial(color);
        meshView.setMaterial(material);
    }
}

```

```

    if(ambient) {
        AmbientLight light = new AmbientLight(Color.WHITE);
        light.getScope().add(meshView);
        pyramidGroup.getChildren().add(light);
    }
    if(fill) {
        meshView.setDrawMode(DrawMode.FILL);
    }
    //End Step 2g
    return pyramidGroup;
}

```

The new method called `buildPyramid()` converts a height, representing the distance from the base to the pyramid apex, and hypotenuse, the distance of the diagonal of the base, to vertices. It also builds a `TriangleMesh` object. The remaining parameters will allow for customizing how the pyramid is rendered and will be explained later in the chapter. The first step is to declare a new `TriangleMesh` object. If you already had your points, faces, and texture coordinates defined from another process (or model file), you could pass this into an alternative `TriangleMesh` constructor. For this example, define the points using:

```
mesh.getPoints().addAll(float x, float y, float z)
```

The `addAll()` method adds float *triplets*, which represent a point in the scene coordinate system. The order by which these points are added matters as their indices are used to “wind” the triangle faces later. Before winding the triangles, you must first define the 2D texture coordinates that will be associated with each triangle face. This can get complicated to explain and likely warrants a separate example. Fortunately, the texture coordinates are only needed if you intend to wrap your 3D `TriangleMesh` with an image texture file. This example creates a single placeholder coordinate, which will be used to fill out the faces using:

```
mesh.getTexCoords().addAll(float x, float y);
```

Finally, create your triangles by winding the points defined earlier. Taken from Listing 11-10:

```

mesh.getFaces().addAll( //use dummy texCoords
    0,0,2,0,1,0, // Vertical Faces "wind" counter clockwise
    0,0,1,0,3,0, // Vertical Faces "wind" counter clockwise
    0,0,3,0,4,0, // Vertical Faces "wind" counter clockwise
    0,0,4,0,2,0, // Vertical Faces "wind" counter clockwise
    4,0,1,0,2,0, // Base Triangle 1 "wind" clockwise because camera has rotated
    4,0,3,0,1,0, // Base Triangle 2 "wind" clockwise because camera has rotated
); //End Step 2d

```

The `getFaces().addAll()` accepts integers that represent a composite of predefined points and texture coordinates. Figure 11-7 breaks down the first face from the example, demonstrating the winding of the points in a counter-clockwise fashion.

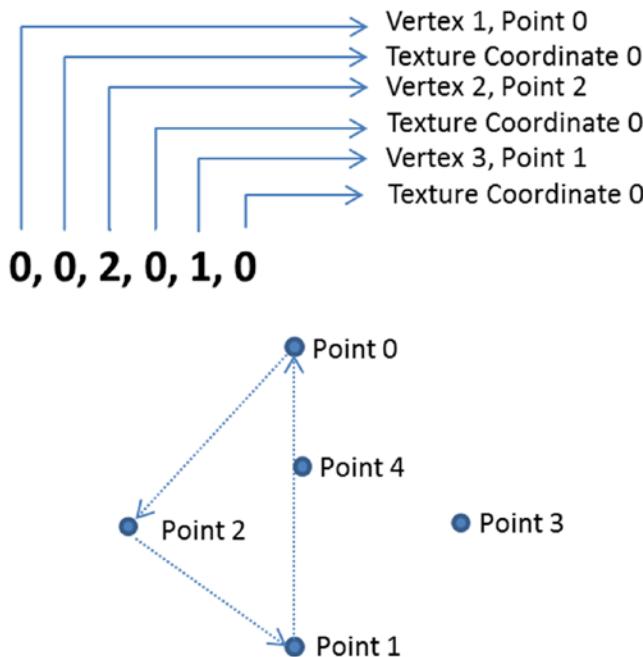


Figure 11-7. Triangle winding using predefined pyramid points

The counter-clockwise winding order tells the camera system that the “front” of the face is facing up, or in this case, facing out from the pyramid object. To build the square base, you again use triangles, using two, but the winding is done clockwise. This is because the upward faces of these triangles are actually facing inward toward the pyramid. By winding clockwise, it tells the camera to effectively render the faces in a flipped fashion. This will give the illusion of a solid geometric object.

MeshViews and DrawMode

TriangleMesh objects by themselves cannot be added and rendered by themselves, as their geometry must be added to a viewable Scenegraph node object. JavaFX 8 provides this via the MeshView class. From Listing 11-10:

```
MeshView meshView = new MeshView(mesh);
```

The MeshView class not only displays the TriangleMesh added to it but allows you to customize lights, fill, materials, and other features. This is demonstrated in Listing 11-10. Let’s use the new buildPyramid() method to exercise the flexibility of the MeshView class. Listing 11-11 uses buildPyramid() to create a pyramid with a GoldenRod colored PhongMaterial and a DrawMode set to LINE.

Listing 11-11. Creating a Single Pyramid

```
/* From TriangleMeshes.java */
```

```
//Step 3a: Create a Pyramid with a color and add to the Scene
//Lets assume our height is 100 and our width (hypotenuse) at the base is 50.
Group pyramid1 = buildPyramid(100,200,Color.GOLDENROD,false,false);
Group pyramidGroup = new Group(pyramid1);
sceneRoot.getChildren().addAll(pyramidGroup);
```

This will render a pyramid similar to Figure 11-8. Most of the `pyramid1` object is *not viewable* because the default scene lighting is positioned in a manner that doesn't reflect much of the geometry.



Figure 11-8. TriangleMesh with default light

To see the entire geometry regardless of light system, turn on ambient lighting:

```
Group pyramid1 = buildPyramid(100,200,Color.GOLDENROD,true,false);
```

Let's also translate the entire geometry closer to the camera.

```
pyramid1.setTranslateZ(-100);
```

You will get a view similar to Figure 11-9. The geometry you defined in your `TriangleMesh` is now clearly visible.

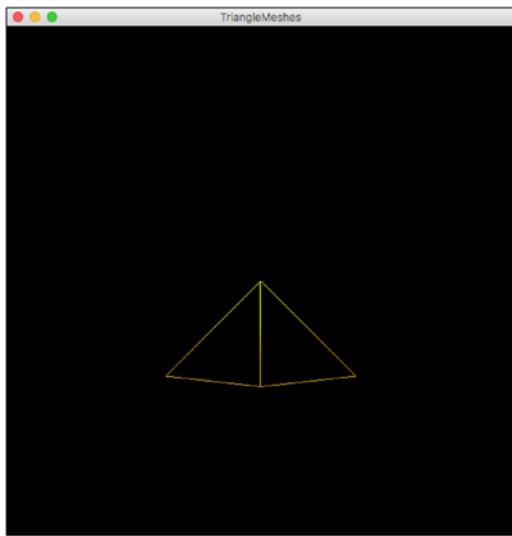


Figure 11-9. TriangleMesh with ambient light

Placing `TriangleMesh` objects within `MeshView` objects and `Group` containers makes basic transforms simple. A rotation used for a primitive:

```
pyramid1.setRotationAxis(Rotate.Y_AXIS);
pyramid1.setRotate(45);
```

After the 45-degree rotation around the Y-axis, it should be similar to Figure 11-10.

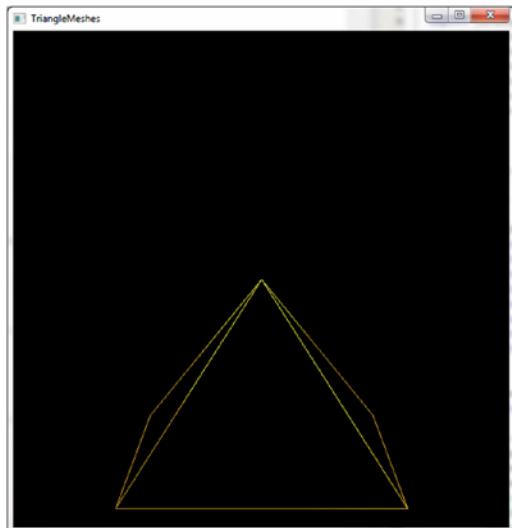


Figure 11-10. TriangleMesh rotated

To fill in the pyramid with colored sides, use the fill parameter in the `buildPyramid()` method:

```
Group pyramid1 = buildPyramid(100,200,Color.GOLDENROD, true, true);
```

This will provide a view similar to Figure 11-11.

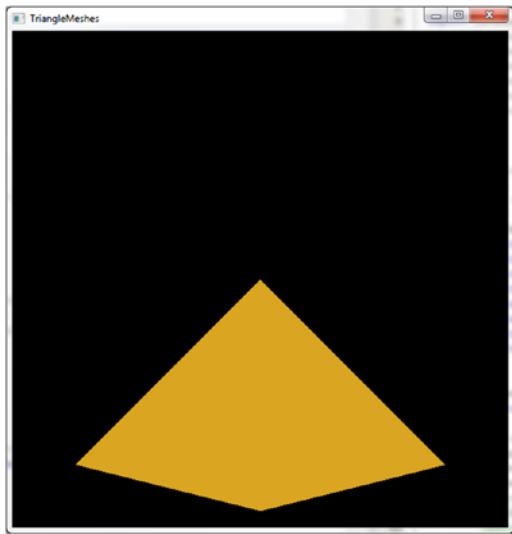


Figure 11-11. TriangleMesh with DrawMode.FILL

Listing 11-12 exercises different combinations of transformations to show the utility of the new `buildPyramid()` method.

Listing 11-12. Creating and Transforming a Pyramid Using DrawMode.FILL

```
/* From TriangleMeshes.java */

//Step 3b: Create and transform a Pyramid using DrawMode FILL
Group pyramid2 = buildPyramid(100,200,Color.GOLDENROD,true,true);
//Since the pyramid is a group it can be translated and rotated like a primitive
pyramid2.setTranslateX(-100);
pyramid2.setTranslateY(-100);
pyramid2.setRotationAxis(Rotate.Z_AXIS);
pyramid2.setRotate(180);
Group pyramidGroup = new Group(pyramid1,pyramid2);
sceneRoot.getChildren().addAll(pyramidGroup);
```

Listing 11-13 exercises the ability of the `buildPyramid()` method to alter the appearance of each pyramid by adding pyramids of different material colors.

Listing 11-13. Creating and Transforming Pyramids of Different Colors.

```
/* From TriangleMeshes.java */
```

```
//Step 3c: Add some more pyramids of a different color
Group pyramid3 = buildPyramid(100,200,Color.LAWNGREEN,true,true);
pyramid3.setTranslateX(100);
Group pyramid4 = buildPyramid(100,200,Color.LAWNGREEN,true,false);
pyramid4.setTranslateX(100);
pyramid4.setTranslateY(-100);
pyramid4.setRotationAxis(Rotate.Z_AXIS);
pyramid4.setRotate(180);
Group pyramidGroup = new Group(pyramid1,pyramid2,pyramid3,pyramid4);
sceneRoot.getChildren().addAll(pyramidGroup);
```

Listings 11-12 and 11-13 clearly demonstrate the flexibility of the `TriangleMesh` and `MeshView` classes within a 3D scene. Adding this code to the example will provide a view similar to Figure 11-12.

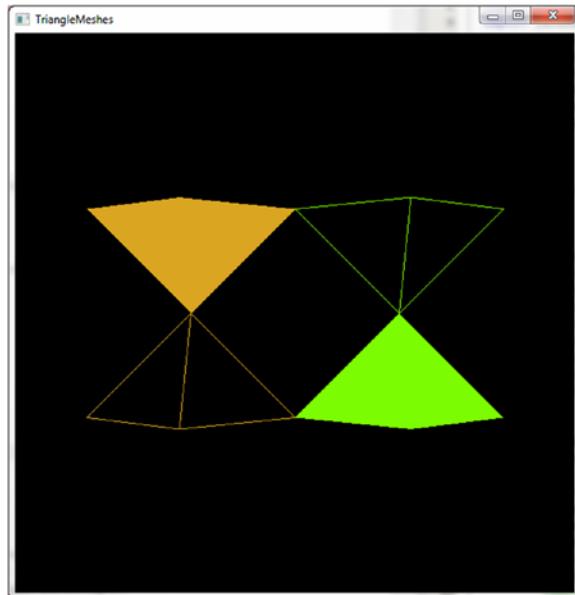


Figure 11-12. Multiple pyramids customized and transformed

Roll Camera!

The new pyramids still look a little flat. You need a way to roll the camera around the pyramids as if it were a satellite orbiting a point of origin. In fact, Listing 11-14 treats the `pyramidGroup` as an origin of sorts by adding a simple mouse handler and converting motion to a rotational transform of the `pyramidGroup` itself.

Listing 11-14. Origin Fixed Camera Rotation Mouse Handler

```
/* From TriangleMeshes.java */

//Step 4a: Add a Mouse Handler for Rotations
Rotate xRotate = new Rotate(0, Rotate.X_AXIS);
Rotate yRotate = new Rotate(0, Rotate.Y_AXIS);
pyramidGroup.getTransforms().addAll(xRotate,yRotate);
//Use Binding so your rotation doesn't have to be recreated
xRotate.angleProperty().bind(angleX);
yRotate.angleProperty().bind(angleY);
//Start Tracking mouse movements only when a button is pressed
scene.setOnMousePressed(event -> {
    scenex = event.getSceneX();
    sceney = event.getSceneY();
    anchorAngleX = angleX.get();
    anchorAngleY = angleY.get();
});
//Angle calculation will only change when the button has been pressed
scene.setOnMouseDragged(event -> {
    angleX.set(anchorAngleX - (scenex - event.getSceneY()));
    angleY.set(anchorAngleY + sceney - event.getSceneX());
});
```

This camera control is different from the earlier example in that it does not actually move or rotate the camera node but the objects in the scene. This gives the illusion that the viewer and camera is rotating around the scene. This is an effective means for viewing scenes that are intending to be facing in. The trick is to use JavaFX binding to update an existing set of `Rotate` objects. This saves the event handler from having to repeatedly create and replace various transformation objects. By adding the handler to the example and giving it a spin, you can achieve viewing angles similar to Figure 11-13.

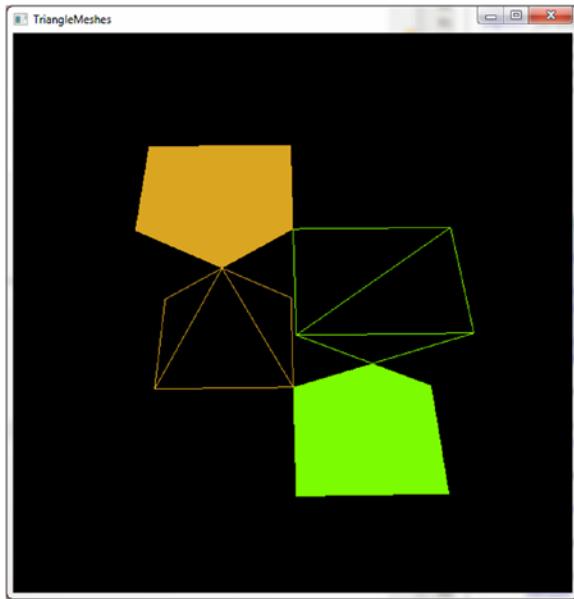


Figure 11-13. Rotating the camera around a fixed origin

The view showing in Figure 11-13 in particular is good because it shows the various pyramids in their different forms. Of particular note is the bottom of the base of the pyramid, which is made up of two triangles using clockwise winding so that their “down” faces are facing outward toward the camera.

Hit the Lights

We have come so far with this custom Pyramid example, yet the 3D geometries still look a little cartoonish. This is because of the lighting, specifically the scene lighting. To truly show off the 3D surfaces, you must add a *point light* to your scene. Listing 11-15 adds a simple point light to the example scene.

Listing 11-15. Adding a Point Light to a Scene

```
/* From TriangleMeshes.java */

//Step 4b: Add a Point light to show specular highlights
PointLight light = new PointLight(Color.WHITE);
sceneRoot.getChildren().add(light);
light.setTranslateZ(-sceneWidth/2);
light.setTranslateY(-sceneHeight/2);
```

Listing 11-15 includes some basic translations to move the light to position other than the default camera position, thereby creating some nice specular highlights on the PhongMaterials of each filled Pyramid. Adding Listing 11-15 to the example and rotating the camera provides nice views such as the one shown in Figure 11-14.

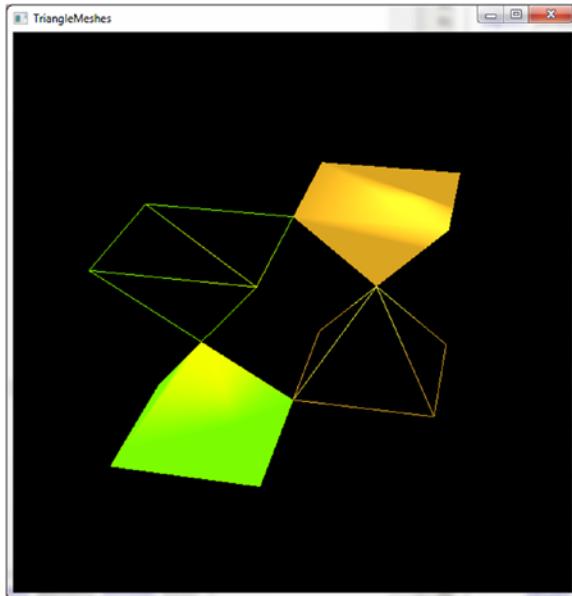


Figure 11-14. PointLight effects on various pyramid TriangleMesh objects

Summary

As mentioned at the beginning of this chapter, a separate book could be dedicated to discussing the JavaFX 9 3D support. There are many features that were alluded to but not discussed, as they would warrant whole chapters. Whole texts are dedicated to the math and abstract concepts of 3D software development and so these things were not discussed in detail.

Hopefully, this chapter scratched the surface of what is possible and hopefully scratched your itch to learn about the new 3D support provided by JavaFX 9. The chapter includes two separate running examples to discuss the usage of primitives, camera nodes, lights, and custom geometries and triangle meshes. Both examples featured event-driven interaction with different approaches to manipulating either the camera or the scene objects with the keyboard and mouse. You were introduced to a list of classes, some dedicated to 3D support, others versatile enough to be applied to either 3D or 2D scenes. Very useful and practical 3D tools can be developed from the building blocks provided in this chapter.

CHAPTER 12



JavaFX and Arduino

Arduino, an open source electronics prototyping platform, started in 2003, so it has been here since the very beginning of the “Internet of Things” (IoT) concept as we understand it now. In fact, the advent of low-cost prototyping platforms like Arduino is one of the cornerstones that have allowed this emerging revolution to happen, supporting both the do-it-yourself concept and the maker movement.

This chapter gives you a quick glance at how you can use JavaFX along with an Arduino board to develop desktop applications for monitoring data coming from the real world or controlling real devices.

It starts by presenting briefly the available boards, followed by the software required to program the microcontroller and the procedure to connect it to a computer (without any WiFi or Ethernet shield solution); finally, you’ll examine the code to acquire the data stream and represent the data and control the devices on the JavaFX thread.

The Arduino Board

Arduino is an open source board with one microcontroller. The best-known version, the Arduino Uno rev 3, is based on the ATmega328 8-bit Atmel AVR microcontroller, while the new Arduino Due, based on the Atmel SAM3X8E ARM Cortex-M3 CPU, is the first Arduino board based on a 32-bit ARM core.

Thanks to a preprogrammed boot loader, programs can be easily uploaded into the microcontroller memory, directly from a connected computer.

The Uno provides 14 digital I/O pins and 6 analog inputs, as female headers, so plug-in shields can be mounted on top of them, or simple jumpers can be plugged to required pins. 32KB are available as flash memory, and the clock operates at 16MHz. Further details can be checked here: <http://arduino.cc/en/Main/ArduinoBoardUno>. Figure 12-1 shows front and back views of the Uno board.



Figure 12-1. Arduino Uno (front and back)

The Due, shown in Figure 12-2, increases the capacity to 54 digital I/O pins and 14 analog pins (12 inputs, 2 outputs). Flash memory is now 512KB, while clock speed is 84MHz. Further details can be checked at <http://arduino.cc/en/Main/ArduinoBoardDue>.

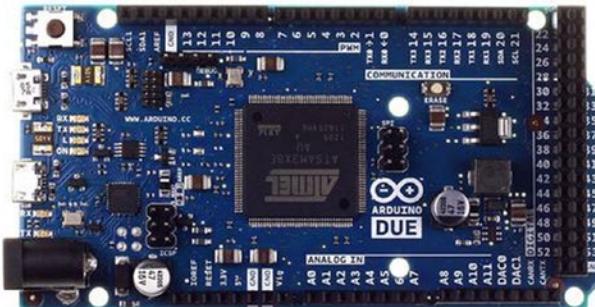


Figure 12-2. Arduino Due (front)

There are other boards with different specifications or components. For any concrete application, you should acquire the one that suits your needs, carefully reading the specifications at <http://arduino.cc/en/Main/Products>.

But it is worth mentioning that a new interesting board is available since 2016: the Arduino 101 (USA)/Genuino 101 (outside the US). Shown in Figure 12-3, it was designed in collaboration with Intel, and includes the Intel® Curie module, a low-power processor based on the Intel Quark SE system-on-a-chip that includes motion sensor, Bluetooth, and battery-charging features. While it keeps the same form factor, the same peripheral list of the Uno and the same entry level price, it comes with the significant addition of onboard Bluetooth LE capabilities and a six-axis accelerometer and gyroscope.

Further details can be checked at <https://www.arduino.cc/en/Main/ArduinoBoard101>.

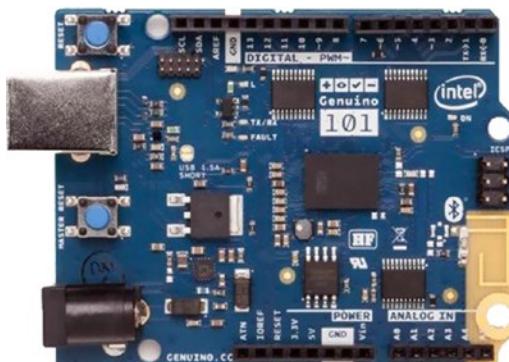


Figure 12-3. Arduino 101/Genuino 101 (front)

For the example in this chapter, we'll use a Genuino 101 board. If you don't have one, there are plenty of sources where you may buy one online, as you can see at this link: <https://store.arduino.cc>.

Once you have the Arduino board and it's using a standard USB cable (type A plug to type B plug), you can attach it to your computer, as shown in Figure 12-4. This connection will provide 5V to power the board (check that the green power LED is illuminated) and a way to program the microcontroller from your computer.

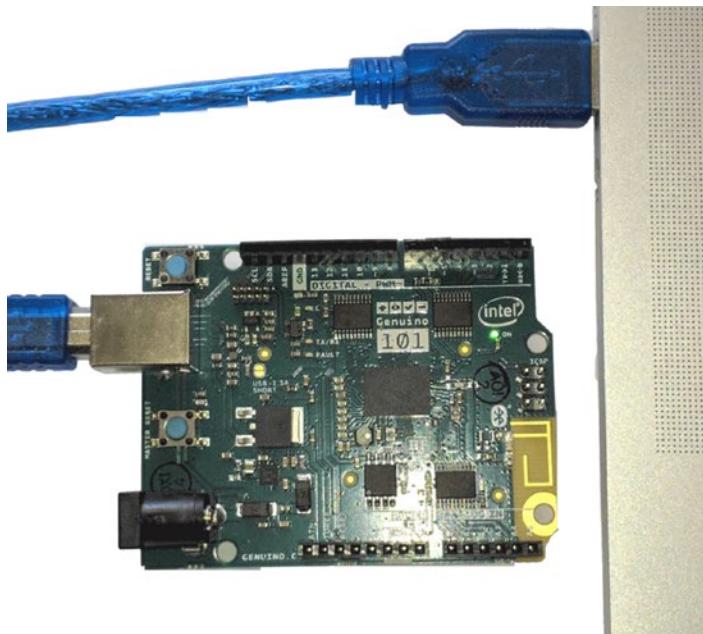


Figure 12-4. Arduino connected to the computer

Later on, after you have programmed the microcontroller, you can use an external power supply, like a 9V battery or a 9-to-12V DC adapter, providing 250mA up to 1000mA, depending on the devices connected, by plugging a 2.1mm plug center pin positive into the board's power jack.

Programming the Arduino

Anyone who knows about Arduino should be familiar with its Arduino IDE, a desktop open source application used to write code and upload it to the board, which runs on every platform. The IDE, written in Java, is based on processing and other open source software.

Recently, there is an online alternative: the new Arduino Web Editor is accessible from any browser, and you can code your sketches online, save them in the cloud, and always have the most up-to-date version of the IDE, including all the contributed libraries and support for new Arduino boards.

Arduino Web Editor

The Arduino Web Editor, which requires a one-time installation plug-in, allows you to upload sketches from the browser onto your boards via an USB cable or the network. It requires a few simple steps to set up this plug-in:

1. Access the Arduino Create portal at <https://create.arduino.cc/getting-started> and click the Setup the Arduino Editor Plugin link, as shown in Figure 12-5.

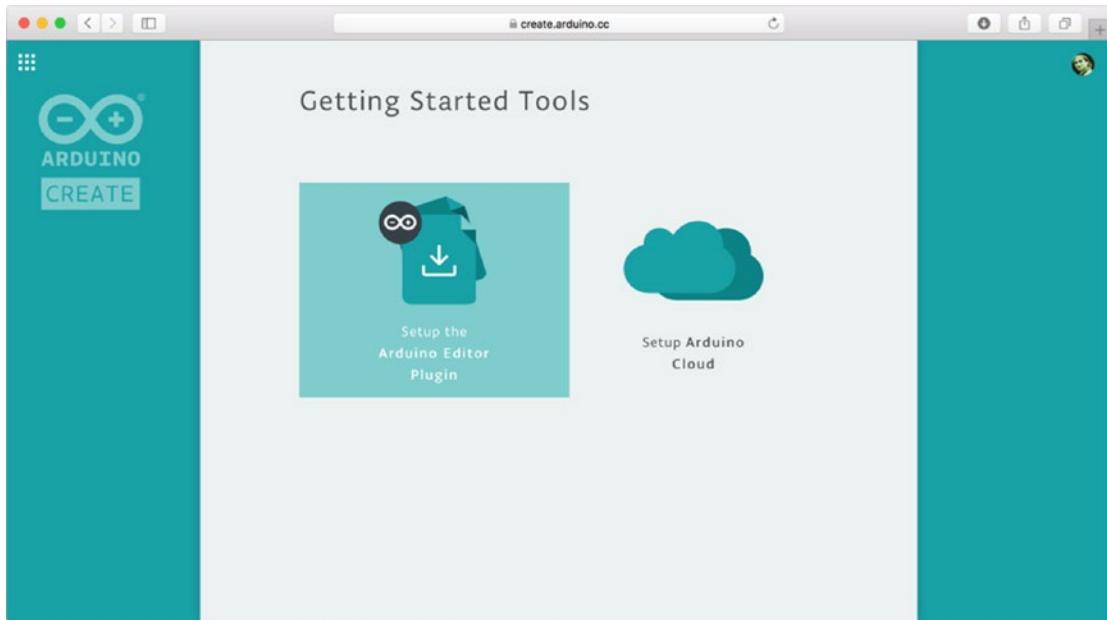


Figure 12-5. Arduino Editor plug-in setup

2. Click Next and download the plug-in on your computer, depending on your platform: <http://create.arduino.cc/getting-started/plugin?page=2>. See Figure 12-6.



Figure 12-6. Download the Arduino plug-in

3. Once it's downloaded, double-click on the installer to start the installation flow and follow the steps in the installer flow.
4. At the end of the process, you should see the Arduino plug-in icon on the tray icon (Windows) or on the menu bar of your Mac. Click on the tray icon for a link to the Arduino Web Editor, or if you want to pause the plug-in, as shown in Figure 12-7.

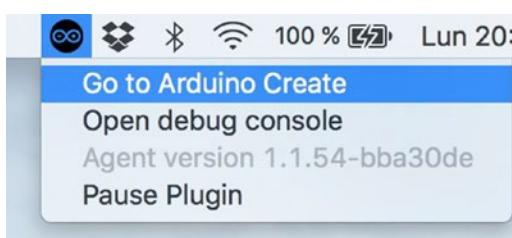


Figure 12-7. Tray icon on the Mac

5. The Arduino Web Editor will open in your default browser, as shown in Figure 12-8. It is required that you sign up and create an account, so you can keep track of the different sketches you create, as well as to access other online features.

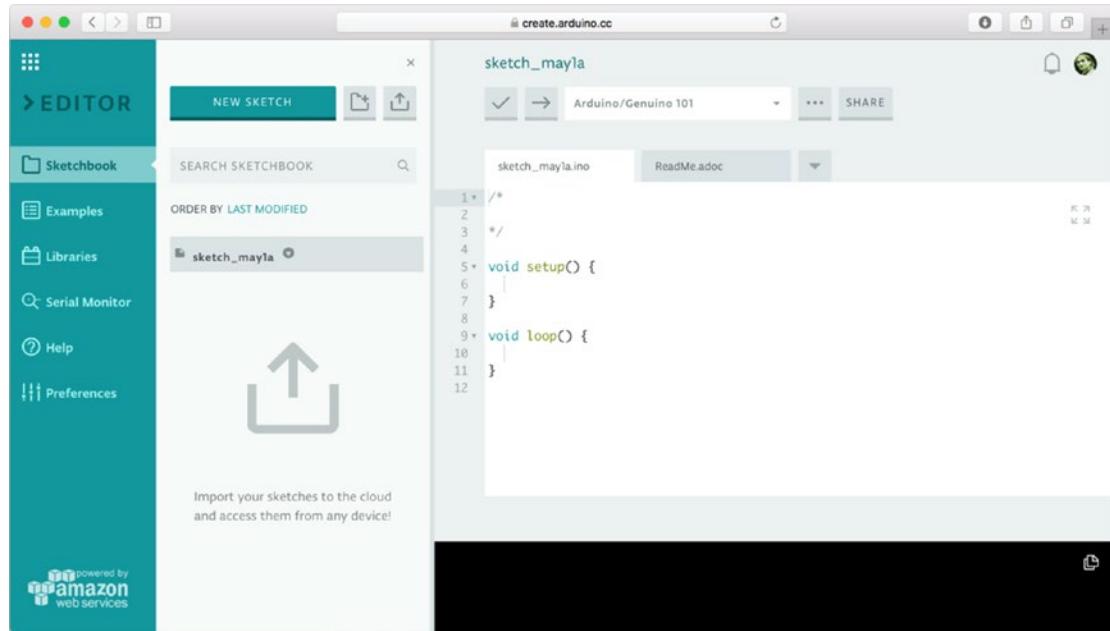


Figure 12-8. Arduino Web Editor

Arduino IDE

You can program the micro using the classic desktop IDE as well. For that, you need to install the drivers and the proper software first. Go to the Downloads section at <http://arduino.cc/en/Main/Software>, select your OS, and download the latest release, which is 1.8.2 at the moment of this writing. Note that for the Arduino/Genuino 101 board, you'll need at least version 1.6.7.

Windows

Download the Windows installer, then double-click on it and proceed with the two steps shown in Figure 12-9.

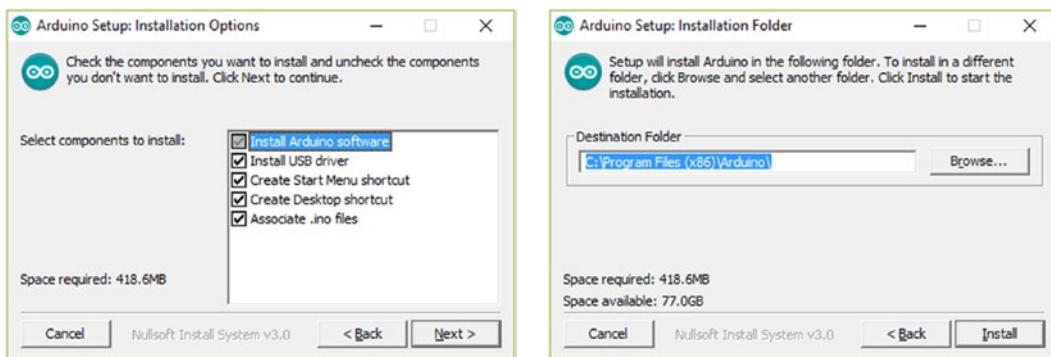


Figure 12-9. Installing Arduino IDE on Windows

At the end of the process, you'll be asked to install the driver. At this point, connect the Arduino to your computer USB port, and a COM connection will be established. A driver should automatically be installed for it. Check the Windows Device Manager to confirm that under Ports you see your Arduino board connected—Expand Ports (COM & LPT)—and you should see a COM port, which will be your Arduino, as shown in Figure 12-10 (in this case, COM3).

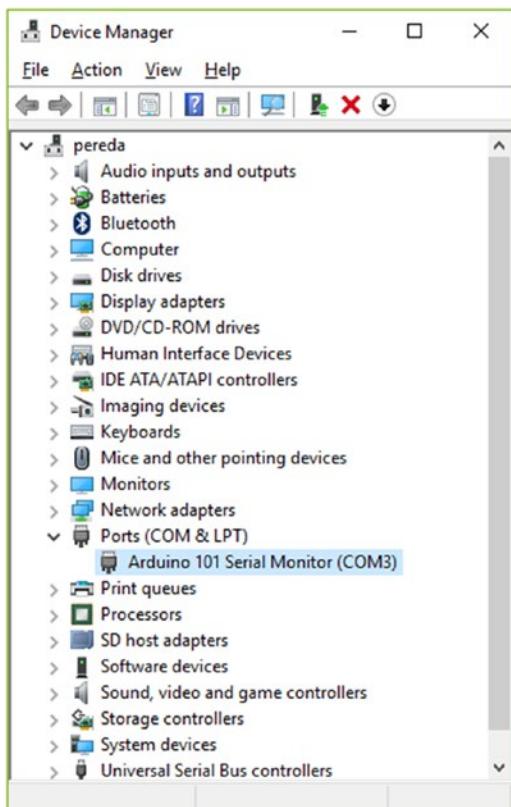


Figure 12-10. Arduino serial port in Windows

If you find a yellow triangle over Arduino under Other Devices, you will need to install the driver manually. Right-click under Arduino and select Update Driver Software. Select Browse My Computer for Driver Software and find the folder called Drivers under the Arduino installation folder. Click Next.

You will be prompted with a security window; select Install. After a few seconds, a success window will inform you the driver has been installed.

MacOS X or Linux

On MacOS X, download the file and copy it to your Applications folder. No drivers are required. When you connect the Arduino board, you should see it listed under /dev/tty.usbmodemXXXX or /dev/tty.usbserialXXXX.

On Debian distributions (for example, Raspberry Pi), you can simply install it on the terminal:

```
$ sudo apt-get install arduino
```

For others distributions, you can download the file for either 32 or 64 bits, unpack it, and install the required dependencies.

When you connect the Arduino board, you should see it listed under /dev/ttyACMX or /dev/ttyUSBX.

Running the IDE

Assuming your installation ends successfully, launch the application; you'll see a screen like Figure 12-11.

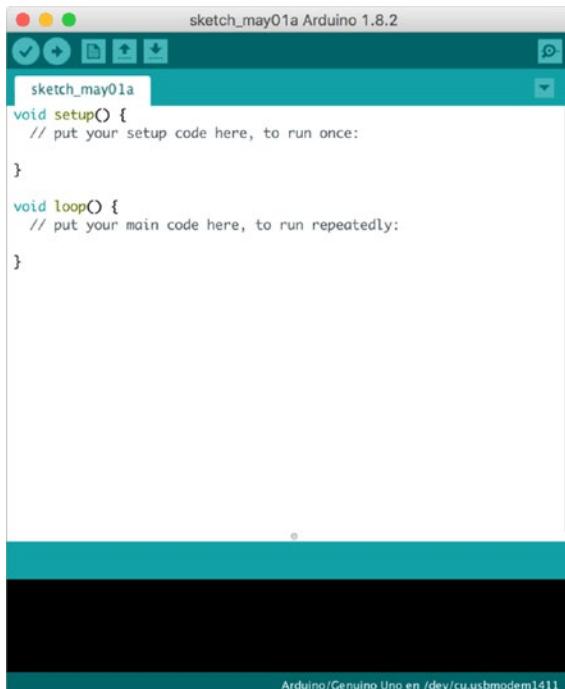


Figure 12-11. The Arduino IDE

The first thing to do is choose Tools ► Board and select the right board from the list. Arduino/Genuino Uno is referred to as the classic Arduino Uno board. If you plug your Arduino/Genuino 101 in for the first time, the Arduino IDE will ask you to install some package to be able to use the board, as shown in Figure 12-12.

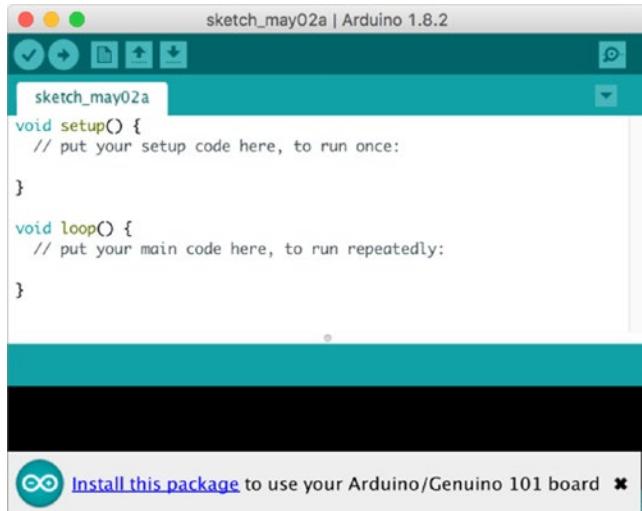


Figure 12-12. Library required for 101 board

Click on the link and the Boards Manager will show up. Under Intel Curie Boards by Intel, click on More Info and press the Install button, as shown in Figure 12-13. After a few minutes, the package will be installed. Then make sure you select the proper board from Tools ► Boards ► Arduino/Genuino 101.

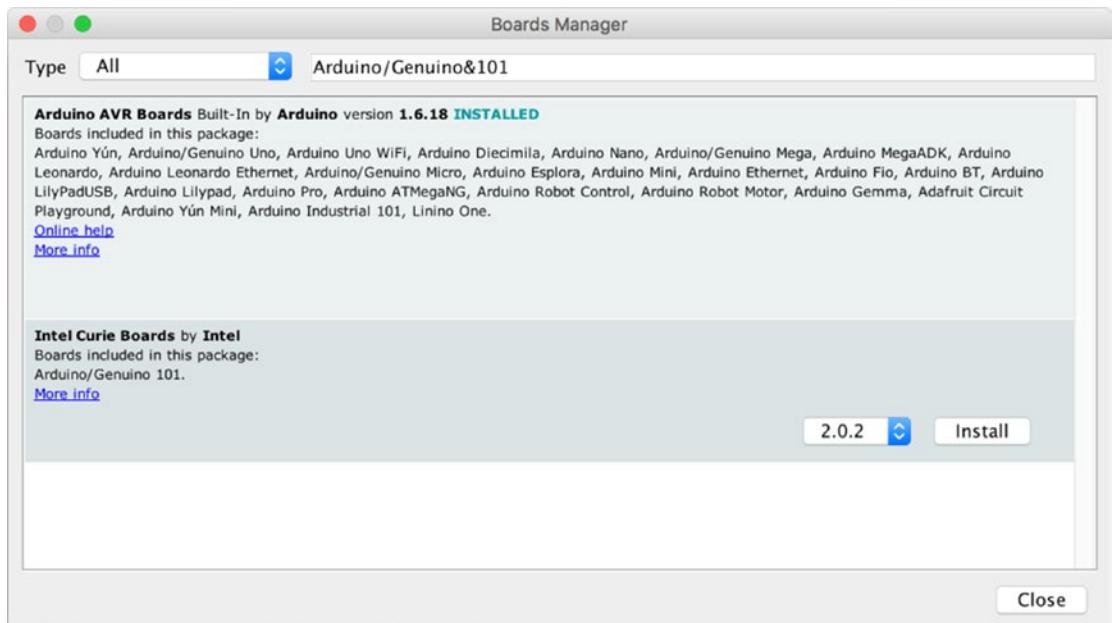


Figure 12-13. Installing the Intel Curie Boards package

Under Serial Port, select the port where the board is connected (i.e., COM3 (Arduino/Genuino 101)). When there are several options, an easy way to find out which port is the right one is to unplug the board, look for the missing port, plug it in again, and select that serial port.

The Blink Example

The quickest test you can do, even without sensors (instrumentation or analog/digital signaling devices), is to open the Blink sample (choose File > Examples > 01.Basics > Blink), as shown in Figure 12-14. The Arduino code files, called sketches, are saved with the extension .ino, under the Arduino/examples folders in several categories.

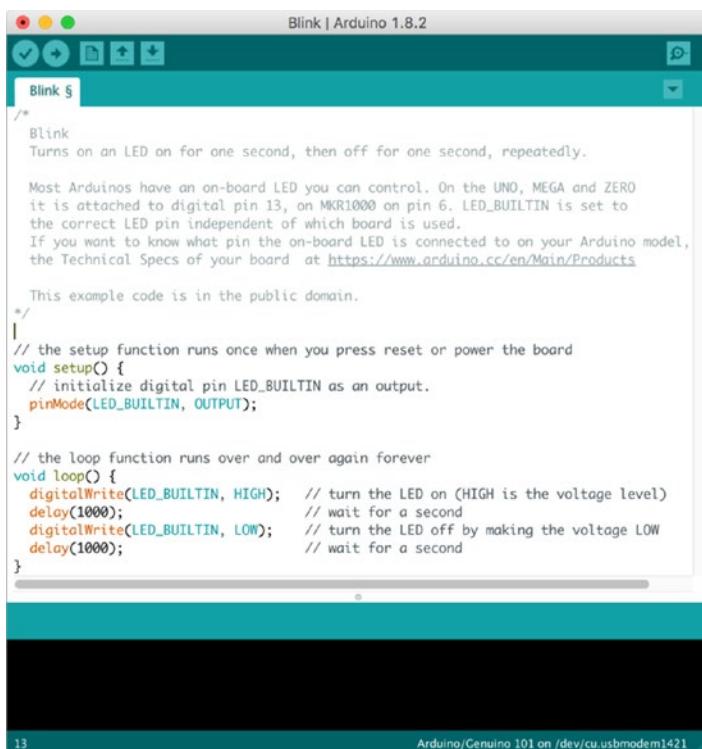


Figure 12-14. The Blink sketch (IDE version)

Click the Upload button (second from the left on the toolbar) and wait for the “Done uploading” message on the status bar. Then you should see the LED labeled with an L blinking on the board.

If you are running the Web Editor version, select the Blink sample from the left menu (Examples ► 01.Basics ► Blink), as shown in Figure 12-15. Make sure your board is listed on the top-right drop-down list and click the Upload button (the right arrow).

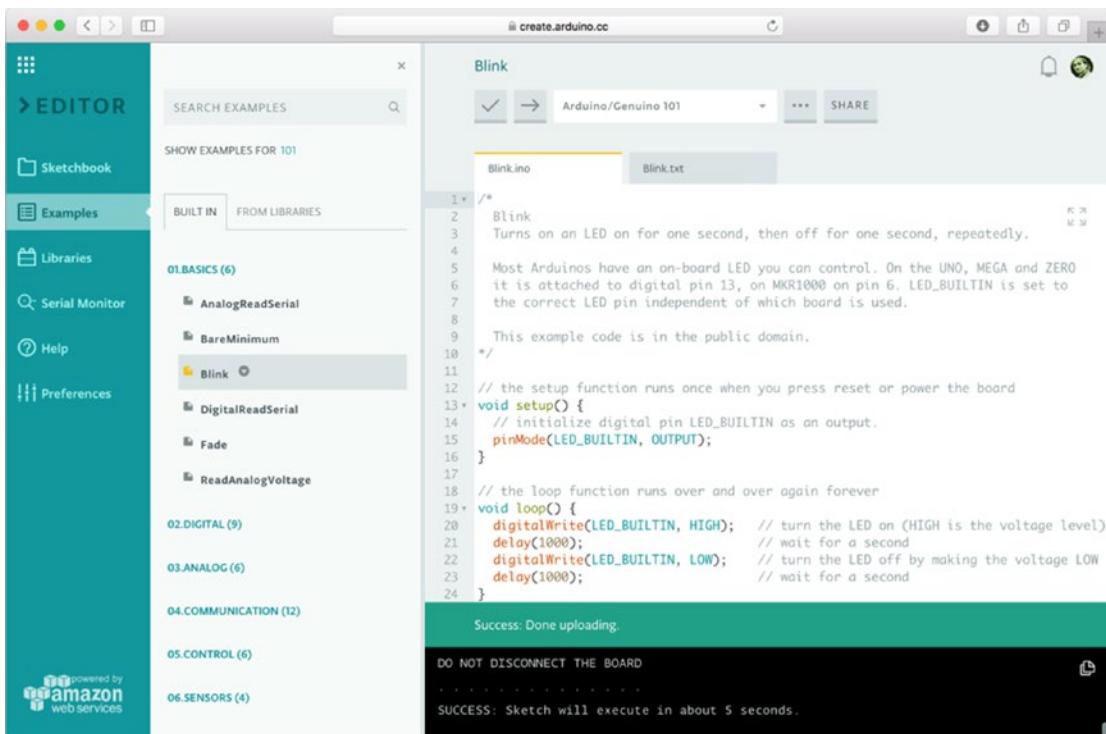


Figure 12-15. The Blink sketch (Web Editor version)

A really good source for example code to start playing with your Arduino board is at the Arduino Playground: <http://playground.arduino.cc/>. You can also visit the Project hub site at <https://create.arduino.cc/projecthub/> for a big collection of projects from the community.

To try most of the preliminary tutorials or your own experiments, you'll need a basic set of sensors and a small set of electronic components. It is a good idea to acquire your Arduino board bundled with electronic components as a starter kit. In addition to the board, you will want a USB cable, a breadboard, jumper wires, LEDs of several colors, resistors, a temperature sensor, a potentiometer, push buttons, and the like.

Orientation Visualizer Example

This example demonstrates how to use Genuino 101's on-board six-axis accelerometer/gyro to read the X, Y, and Z values of both the accelerometer and the gyroscope. The accelerometer values are used to determine the orientation of the board, while the gyroscope measures the angular velocity of the board. The example is based on this link: <https://www.arduino.cc/en/Tutorial/Genuino101CurieIMUOrientationVisualiser>.

The only hardware required is the Arduino/Genuino 101 board.

The software required for the example is the Madgwick library, available from the Library Manager. Using the the Arduino IDE, go to Sketch ▶ Include Library ▶ Manage Libraries. There you can search for "Madgwick" and install the latest version of the library directly from there, as shown in Figure 12-16. The source code of the library can be found at <https://github.com/arduino-libraries/MadgwickAHRS>.

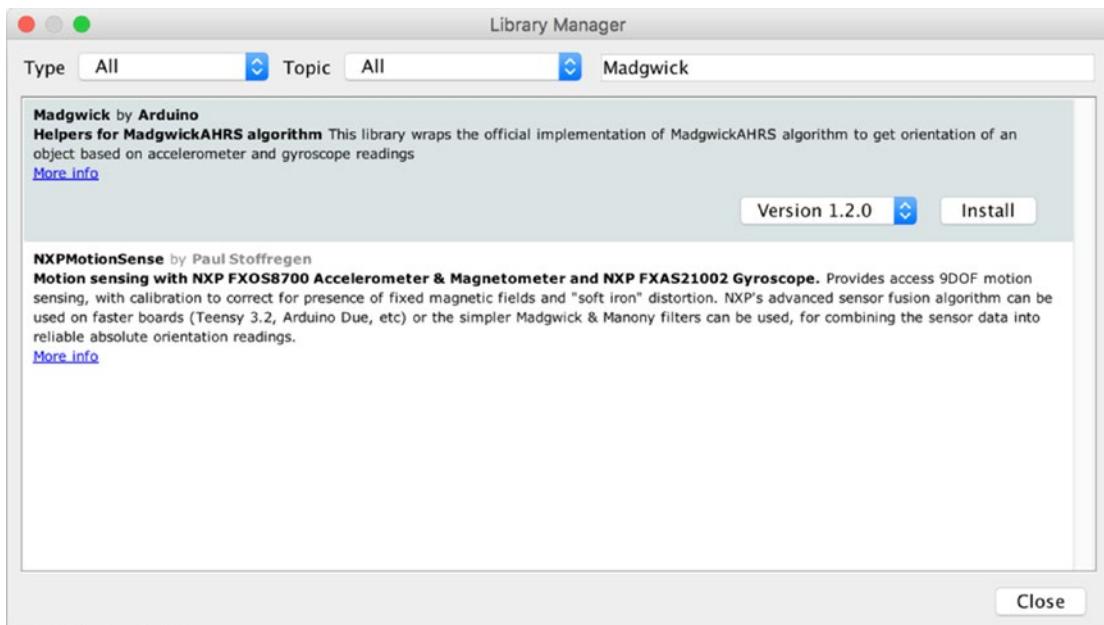


Figure 12-16. Installing the Madgwick library

Now open the Visualize101.ino sample from the library folder that you will find under <User>/Documents/Arduino/Libraries/Madgwick/examples/Visualize101. You need to modify it, in order to add an initial calibration. Listing 12-1 includes the modified source code.

Listing 12-1. Visualize101 Source Code with Calibration

```
#include <CurieIMU.h>
#include <MadgwickAHRS.h>

Madgwick filter;
unsigned long microsPerReading, microsPrevious;
float accelScale, gyroScale;

void setup() {
    Serial.begin(9600);
    while (!Serial); // wait for the serial port to open
```

```
// start the IMU and filter
CurieIMU.begin();
CurieIMU.setGyroRate(25);
CurieIMU.setAccelerometerRate(25);
filter.begin(25);

// Set the accelerometer range to 2G
CurieIMU.setAccelerometerRange(2);
// Set the gyroscope range to 250 degrees/second
CurieIMU.setGyroRange(250);

// Starting Gyroscope calibration
CurieIMU.autoCalibrateGyroOffset();
// Starting Acceleration calibration...");
CurieIMU.autoCalibrateXAccelOffset(0);
CurieIMU.autoCalibrateYAccelOffset(0);
CurieIMU.autoCalibrateZAccelOffset(1);

// Enabling Gyroscope/Acceleration offset compensation
CurieIMU.setGyroOffsetEnabled(true);
CurieIMU.setAccelOffsetEnabled(true);

// initialize variables to pace updates to correct rate
microsPerReading = 1000000 / 25;
microsPrevious = micros();
}

void loop() {
    int aix, aiy, aiz;
    int gix, giy, giz;
    float ax, ay, az;
    float gx, gy, gz;
    float roll, pitch, heading;
    unsigned long microsNow;

    // check if it's time to read data and update the filter
    microsNow = micros();
    if (microsNow - microsPrevious >= microsPerReading) {

        // read raw data from CurieIMU
        CurieIMU.readMotionSensor(aix, aiy, aiz, gix, giy, giz);

        // convert from raw data to gravity and degrees/second units
        ax = convertRawAcceleration(aix);
        ay = convertRawAcceleration(aiy);
        az = convertRawAcceleration(aiz);
        gx = convertRawGyro(gix);
        gy = convertRawGyro(giy);
        gz = convertRawGyro(giz);
```

```
// update the filter, which computes orientation
filter.updateIMU(gx, gy, gz, ax, ay, az);

// print the heading, pitch and roll
roll = filter.getRoll();
pitch = filter.getPitch();
heading = filter.getYaw();
// Serial.print("Orientation: ");
Serial.print(heading);
Serial.print(" ");
Serial.print(pitch);
Serial.print(" ");
Serial.println(roll);

// increment previous time, so we keep proper pace
microsPrevious = microsPrevious + microsPerReading;
}

}

float convertRawAcceleration(int aRaw) {
    // since we are using 2G range
    // -2g maps to a raw value of -32768
    // +2g maps to a raw value of 32767

    float a = (aRaw * 2.0) / 32768.0;
    return a;
}

float convertRawGyro(int gRaw) {
    // since we are using 250 degrees/seconds range
    // -250 maps to a raw value of -32768
    // +250 maps to a raw value of 32767

    float g = (gRaw * 250.0) / 32768.0;
    return g;
}
```

Running the Web Editor, open the Visualize101 project from Examples ► From Libraries ► Madgwick, and include the calibration, as shown in Figure 12-17.

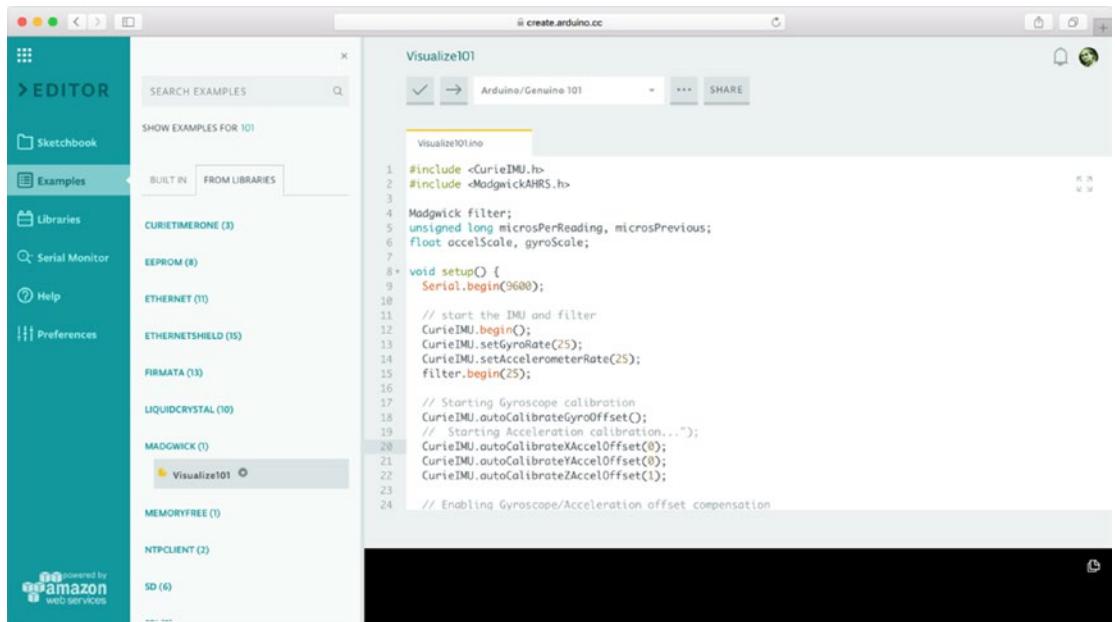


Figure 12-17. Visualize101 in the Web Editor

Make sure you have your board connected and select the right port. Click upload, and after a few seconds the project is compiled and uploaded to the board.

You can open the Serial Monitor to visualize the values of heading (yaw), pitch, and roll (in degrees) printed out, while you hold the board with your hand and wave it, as shown in Figure 12-18.

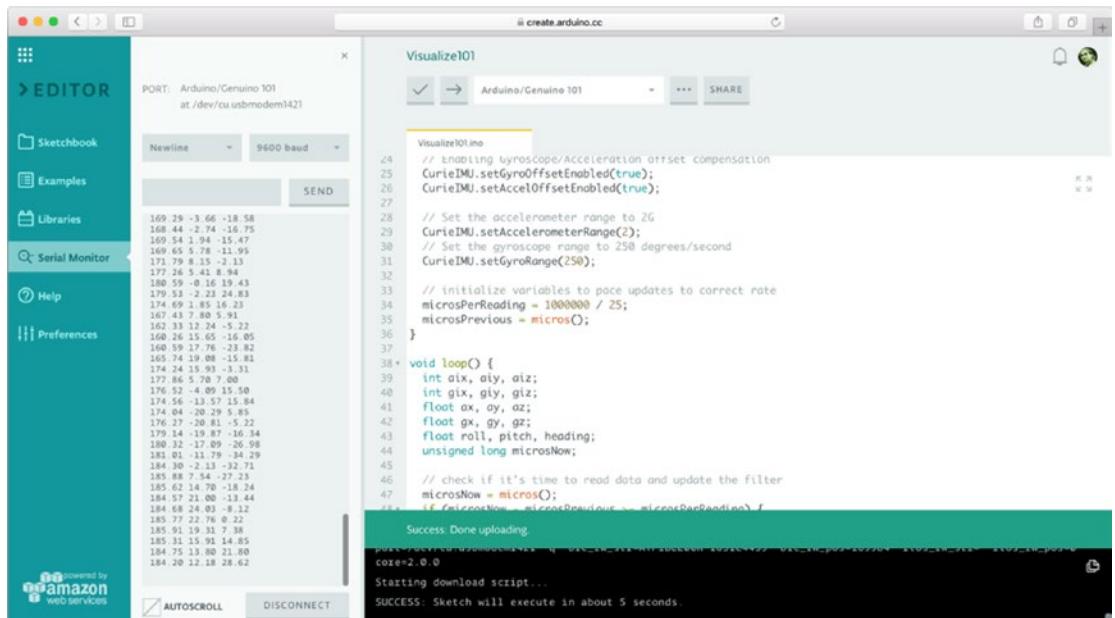


Figure 12-18. Running Visualize101

How It Works

The sketch uses functions inside the CurieIMU library to get the data from the accelerometer/gyroscope. It has two primary methods: `setup()` and `loop()`.

In the `setup()` method, you configure the serial port speed and the CurieIMU library, by setting the sample rate of the accelerometer, gyro, and filter to 25Hz, and the accelerometer range to 2g and the gyro range to 250 °/s. Both gyroscope and accelerometer are calibrated initially. For this, it is recommended you keep the Arduino board steady and in a flat position.

In the `loop()` method, every 25ms you get accelerometer and gyroscope data using the functions from the CurieIMU library and convert the raw data to acceleration (g) and angular velocity (°/s). Using the Madgwick library, those values are converted to the roll, pitch, and yaw angles, which are finally sent to the serial port. These angles are used in many areas, like navigation, to define the spatial orientation of a body.

If everything is working, it's time to start thinking about how you are going to read the serial port to get these measures and how you are going to display those readings in your JavaFX application.

Serial Reading

Although you can find third-party applications for serial reading—like the old HyperTerminal for Windows, Minicom for Linux, and Terminal for Mac—you need to have these readings in your application, and for that, you need a Java API that performs that task.

Note here that serial reading is a highly hardware-specific task, and it breaks the Java multiplatform approach as you need native support for accessing the ports. Besides the old JavaComm API from Sun, there is no official API for communications bundled with the Java SE distribution, so you'll need a third-party library.

For several years, the preferred Java library for serial communication was RXTX. Originally from Trent Jarvi, and distributed under LGPL v 2.1+Linking Over Controlled Interface license, the last version (2.2pre2 from 2010) was accessible at <http://rxtx.qbang.org/>.

But this library was discontinued, and even the Arduino IDE replaced it with a new one—the Java Simple Serial Connector (jSSC)—since it was heavily used for communications between the IDE and the board.

Java Simple Serial Connector

Java Simple Serial Connector (jSSC), from Alexey Sokolov, is licensed under GNU Lesser GPL. The latest version as of this writing (jSSC-2.8.0), released January 2014, was the library chosen to replace RXTX in the Arduino IDE (starting from version 1.5.6).

jSSC is available for all platforms and operating systems. You'll notice it is faster than RXTX, and several bugs have been fixed. jSSC comes in a simple JAR file, with all the native required files included, so you don't need to deal with local installations, as it takes care of adding them to the classpath at runtime.

A JAR with latest available version can be downloaded from <https://github.com/scream3r/java-simple-serial-connector/releases>. If you add it as a library in NetBeans or open it, you will see its classes (the `jssc` package) and the native content (the `libs` package).

JavaFX, the Charting API, and Orientation

In the example that follows, you will learn how to design a simple JavaFX application that takes the orientation readings from the Arduino board and displays them in a chart. For the sake of clarity, the example uses two classes—one for the serial readings in Listing 12-2 and one for the chart and the JavaFX stage in Listing 12-3.

You'll bind these classes (`Serial` and `OrientationFX`) by using a `StringProperty`, containing the last line read from the serial port. By listening to changes in this property in the JavaFX thread, you'll know when there's a new reading to add to the chart.

Creating the Module Project

Let's start by creating a module project in NetBeans, as shown in Figure 12-19. Make sure you select the JDK 9 platform, as in Figure 12-20, when setting the project name, OrientationFX.

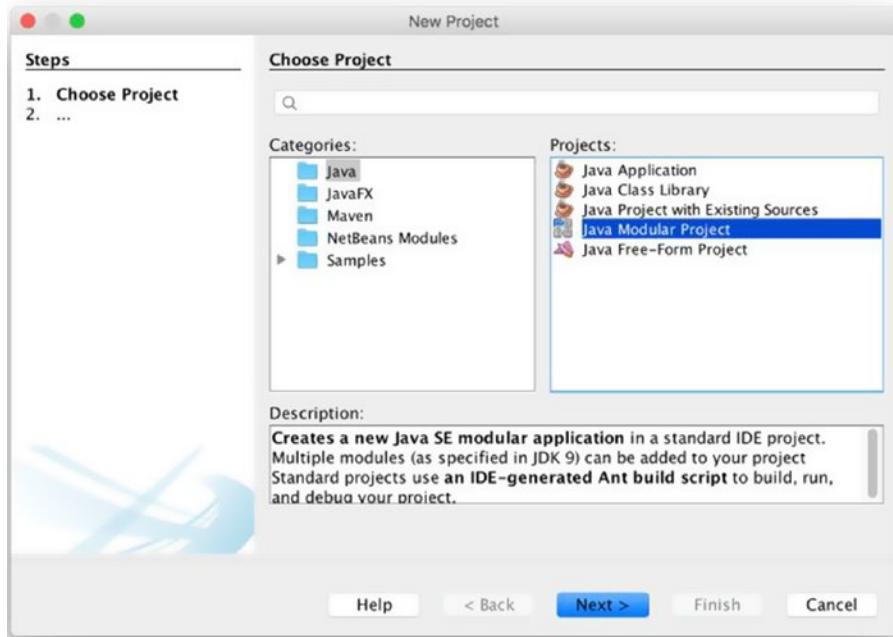


Figure 12-19. Adding a module project in NetBeans

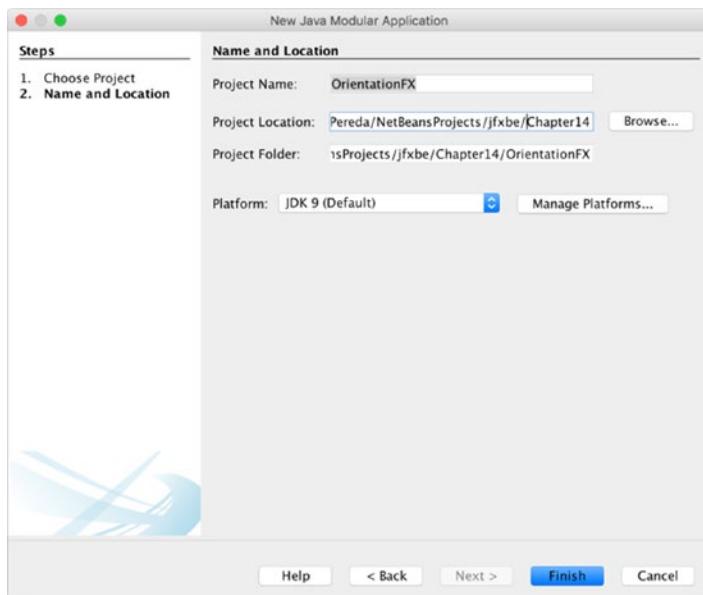


Figure 12-20. Project name and location

Click Finish, and the new empty project will be created. Right-click on it, select New ▶ Module, and name it `com.jfxbe.orientationfx`, as shown in Figure 12-21. Click Finish. The empty class called `module-info.java` will be created.

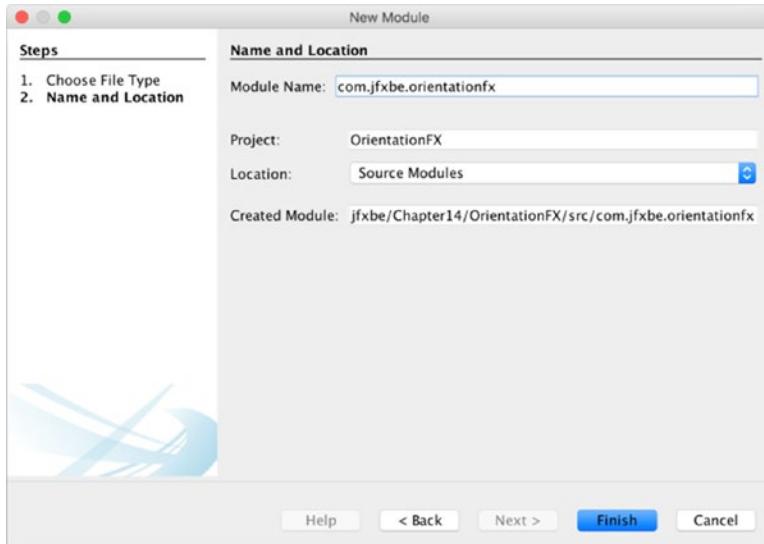


Figure 12-21. Module name

For convenience, place the `jssc.jar` under a `libs` folder in the project, and then add it as a library (choose Libraries ▶ Add JAR/Folder).

Now you can add to the module descriptor the `requires` clause for `javafx.controls`, to include the JavaFX controls and base modules, and the clause for `jssc` as well, as shown in Listing 12-2.

Listing 12-2. Module-info Class

```
module com.jfxbe.orientationfx {
    requires javafx.controls;
    requires jssc;
}
```

You can add now a new package to the `com.jfxbe.orientationfx` module and a new class to it, called `OrientationFX`. The class will extend `Application`. Notice that the class will be available to be imported, thanks to the `requires` clause in the module descriptor. You simply add a `Scene` and show the `stage`, as shown in Listing 12-3.

Listing 12-3. Defining the OrientationFX Class

```
public class OrientationFX extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        Scene scene = new Scene(new StackPane(), 1000, 600);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

Finally, you can add this package to the module with the exports clause, as in Listing 12-4, and run the application to make sure the initial settings are correct. An empty stage should appear.

Listing 12-4. Final Module-info Class

```
module com.jfxbe.orientationfx {
    requires javafx.controls;
    requires jssc;
    exports com.jfxbe.orientationfx;
}
```

Serial Communications

You can add Serial to the package as the class for serial communications. Listing 12-5 contains its source code.

Listing 12-5. The Serial Class Source Code

```
package com.jfxbe.orientationfx;

import java.util.Arrays;
import java.util.List;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import jssc.SerialPort;
import jssc.SerialPortException;
import jssc.SerialPortList;

public class Serial {
    /* List of usual serial ports. Add more or remove those you don't need */
    private static final List<String> USUAL_PORTS = Arrays.asList(
        "/dev/cu.usbmodem1411", "/dev/tty.usbmodem1421", // Mac OS X
        "/dev/usbdev","/dev/ttyUSB","/dev/ttyACM", "/dev/serial", // Linux
        "COM3","COM4","COM5","COM6" // Windows
    );
    private final String ardPort;
    private SerialPort serPort;
```

```

public static final String SEPARATOR = " ";
private static final String LINE_SEPARATOR = "\r\n";
private StringBuilder sb = new StringBuilder();
private final StringProperty line = new SimpleStringProperty("");

public Serial() {
    this("");
}

public Serial(String port) {
    ardPort = port;
}

/* connect() looks for a valid serial port with an Arduino board connected.
 * If it is found, it's opened and a listener is added, so every time
 * a line is returned, the stringProperty is set with that line.
 * For that, a StringBuilder is used to store the chars and extract the line
 * content whenever '\r\n' is found.
 */
public boolean connect(){
    Arrays.asList(SerialPortList.getPortNames())
        .stream()
        .filter(name ->
            ((!ardPort.isEmpty() && name.equals(ardPort)) ||
            (ardPort.isEmpty() &&
            USUAL_PORTS.stream()
                .anyMatch(p -> name.startsWith(p)))))
        .findFirst()
        .ifPresent(name -> {
            try {
                serPort = new SerialPort(name);
                System.out.println("Connecting to " + serPort.getPortName());
                if (serPort.openPort()) {
                    serPort.setParams(SerialPort.BAUDRATE_9600,
                                      SerialPort.DATABITS_8,
                                      SerialPort.STOPBITS_1,
                                      SerialPort.PARITY_NONE);
                    serPort.setEventsMask(SerialPort.MASK_RXCHAR);
                    serPort.addEventListener(event -> {
                        if (event.isRXCHAR()) {
                            try {
                                sb.append(serPort.readString(event.getEventValue()));
                                String ch = sb.toString();
                                if (ch.endsWith(LINE_SEPARATOR)) {
                                    // add timestamp
                                    line.set(Long.toString(System.currentTimeMillis())
                                         .concat(SEPARATOR)
                                         .concat(ch.substring(
                                             0, ch.indexOf(LINE_SEPARATOR)))));
                                    sb = new StringBuilder();
                                }
                            }
                        }
                    });
                }
            }
        });
}

```

```
        } catch (SerialPortException e) {
            System.out.println("Serial error: " + e.toString());
        }
    }
}
} catch (SerialPortException ex) {
    System.out.println("ERROR: Port '" + name + "' : " + ex.toString());
}
});
return serPort != null;
}

public void write(String text) {
try {
    serPort.writeBytes(text.getBytes());
} catch (SerialPortException ex) {
    System.out.println("ERROR: writing '" + text + "' : " + ex.toString());
}
}

public void disconnect() {
if (serPort != null) {
    try {
        serPort.removeEventListener();
        if (serPort.isOpened()) {
            serPort.closePort();
        }
    } catch (SerialPortException ex) {
        System.out.println("ERROR closing port exception: " + ex.toString());
    }
    System.out.println("Disconnecting: comm port closed.");
}
}

public StringProperty getLine() {
    return line;
}

public String getPortName() {
    return serPort != null ? serPort.getPortName() : "";
}
}
```

How It Works

First of all, notice that the jSSC imports are available if you included the `jssc.jar` properly as mentioned. Next, you set a list of suitable port names where the Arduino board may be connected. This list can be extended to other ports in your system or you can delete the ones from other platforms. The `ardPort` variable can be set with a port name through the constructor, and the list won't be used.

```
private static final List<String> USUAL_PORTS = Arrays.asList(
    "/dev/cu.usbmodem1411", "/dev/tty.usbmodem1421", // Mac OS X
    "/dev/usbdev", "/dev/ttyUSB", "/dev/ttyACM", "/dev/serial", // Linux
    "COM3", "COM4", "COM5", "COM6" // Windows
);
private final String ardPort;
public Serial() {
    this("");
}
public Serial(String port) {
    ardPort = port;
}
```

In the `connect()` method, go through the list of valid serial ports found in your system. Use lambdas to filter this collection so you get an item matching the `ardPort` value, if it is set, or one of the `USUAL_PORTS` list, otherwise:

```
public boolean connect(){
    Arrays.asList(SerialPortList.getPortNames())
        .stream()
        .filter(name ->
            ((!ardPort.isEmpty() && name.equals(ardPort)) ||
            (ardPort.isEmpty() &&
            USUAL_PORTS.stream()
                .anyMatch(p -> name.startsWith(p)))))
        .findFirst()
```

If you find a valid serial port, you can try to open it and set several common parameters for the serial communication:

```
.ifPresent(name -> {
    try {
        serPort = new SerialPort(name);
        System.out.println("Connecting to " + serPort.getPortName());
        if (serPort.openPort()) {
            serPort.setParams(SerialPort.BAUDRATE_9600,
                SerialPort.DATABITS_8,
                SerialPort.STOPBITS_1,
                SerialPort.PARITY_NONE);
            serPort.setEventsMask(SerialPort.MASK_RXCHAR);
```

Because you'll be listening to the serial port, you add a listener that reacts to every `SerialPortEvent`, which it detects by the presence of bytes on the input buffer. The number of bytes is determined by `event.getEventValue()`. The `readString(int byteCount)` method returns whenever there are bytes on the buffer. You use a `StringBuilder` to store all the readings, and you evaluate the characters for a carriage return and line feed ("`\r\n`") sequence. If they're found, you extract the line of text and reset the builder. A `StringProperty` is used to store the line, so the JavaFX thread can listen to changes in that observable object (exposed with the `getLine()` method; see Listing 12-6 later in the chapter). You add a timestamp to the string to keep track of the instant it was read. Tracking over time also helps when the measures are the same, so they can be shown sequentially on the chart.

```
serPort.addEventListener(event -> {
    if (event.isRXCHAR()) {
        try {
            sb.append(serPort.readString(event.getEventValue()));
            String ch = sb.toString();
            if (ch.endsWith(LINE_SEPARATOR)) {
                // add timestamp
                line.set(Long.toString(System.currentTimeMillis())
                    .concat(SEPARATOR)
                    .concat(ch.substring(
                        0, ch.indexOf(LINE_SEPARATOR)))));
                sb = new StringBuilder();
            }
        } catch (SerialPortException e) {
            System.out.println("Serial error: " + e.toString());
        }
    }
});
```

Note that in cases where you have to read several measures at a time, you need to use the same separator character in your Arduino sketch, to split them properly into individual readings.

Finally, the `disconnect()` method takes care of removing the listener and closing the port:

```
public void disconnect() {
    if (serPort != null) {
        try {
            serPort.removeEventListener();
            if (serPort.isOpened()) {
                serPort.closePort();
            }
        } catch (SerialPortException ex) {
            System.out.println("ERROR closing port exception: " + ex.toString());
        }
        System.out.println("Disconnecting: comm port closed.");
    }
}
```

Testing Serial Comms

Before getting into the Charts API, let's add a quick test to the Application class, as shown in Listing 12-6. Open the serial port upon starting the application and close it when the app is shut down.

Listing 12-6. Testing Serial Comms

```
public class OrientationFX extends Application {

    private Serial serial;

    @Override
    public void start(Stage primaryStage) throws Exception {
        Scene scene = new Scene(new StackPane(), 800, 600);
        primaryStage.setScene(scene);
        primaryStage.show();

        serial = new Serial();
        serial.getLine().addListener((obs, ov, nv) -> {
            String[] split = nv.split("\\s");
            System.out.println("Orientation " + split[1] + " " + split[2] + " " + split[3]);
        });
        serial.connect();
    }

    @Override
    public void stop() throws Exception {
        if (serial != null) {
            serial.disconnect();
        }
    }
}
```

Run the test to see if you find the orientation values in your console.

```
Connecting to /dev/tty.usbmodem1421
208.50 -1.67 -0.42
208.48 -2.14 -0.20
208.49 -1.75 -0.42
208.46 -2.11 -0.12
208.47 -1.79 -0.43
Disconnecting: comm port closed.
```

The JavaFX Charts API

Charts with two axes (bars, areas, lines, bubble, or scatter) or pie charts without an axis have been available in JavaFX since version 2.0. Each chart is a node, so it can be added to a scene like any other node.

In this example, you'll add a LineChart to plot the three orientation angles. This requires three series to plot the readings, with XYChart.Data as pair of values for every point, one for each axis.

For convenience, whenever the size of the series is greater than 300 points, the first values will be removed. Listing 12-7 shows the `ArduinoChart` class, which creates a container for the chart and processes any possible serial event.

Listing 12-7. The `ArduinoChart` Class

```
/**
 * @author Jose Pereda
 */
public class ArduinoChart {

    private static final int SIZE_MAX = 300;

    private LineChart<Number,Number> chart;
    private Series<Number, Number> rollSeries, pitchSeries, yawSeries;
    private NumberAxis xAxis, yAxis;

    private Label labelRoll, labelPitch, labelYaw;

    public VBox createArduinoChart() {
        xAxis = new NumberAxis();
        xAxis.setLabel("Time");
        xAxis.setAutoRanging(true);
        xAxis.setForceZeroInRange(false);
        xAxis.setTickLabelFormatter(new StringConverter<Number>() {
            @Override
            public String toString(Number t) {
                return new SimpleDateFormat("HH:mm:ss")
                    .format(new Date(t.longValue()));
            }
            @Override
            public Number fromString(String string) {
                throw new UnsupportedOperationException("Not supported yet.");
            }
        });
        yAxis = new NumberAxis();
        yAxis.setLabel("Angle (°)");
        chart = new LineChart<>(xAxis, yAxis);
        chart.setCreateSymbols(true);
        chart.setAnimated(false);
        chart.setLegendVisible(true);
        chart.setTitle("Roll/Pitch/Yaw");
        rollSeries = new Series<>();
        rollSeries.setName("Roll (°)");
        pitchSeries = new Series<>();
        pitchSeries.setName("Pitch (°)");
        yawSeries = new Series<>();
        yawSeries.setName("Yaw (°)");

        chart.getData().addAll(rollSeries, pitchSeries, yawSeries);
    }
}
```

```

labelRoll = new Label();
labelPitch = new Label();
labelYaw = new Label();

final HBox hBox = new HBox(new Label("Values: "), labelRoll, labelPitch, labelYaw);
hBox.getStyleClass().add("box");

final VBox vBox = new VBox(chart, hBox);
vBox.getStyleClass().add("box");
return vBox;
}

public void processEvent(long time, double roll, double pitch, double yaw) {
    rollSeries.getData().add(new Data(time, roll));
    labelRoll.setText("Roll: " + String.format("%3.2f °", roll));
    if (rollSeries.getData().size() > SIZE_MAX) {
        rollSeries.getData().remove(0);
    }

    pitchSeries.getData().add(new Data(time, pitch));
    labelPitch.setText("Pitch: " + String.format("%3.2f °", pitch));
    if (pitchSeries.getData().size() > SIZE_MAX) {
        pitchSeries.getData().remove(0);
    }

    yawSeries.getData().add(new Data(time, yaw));
    labelYaw.setText("Yaw: " + String.format("%3.2f °", yaw));
    if (yawSeries.getData().size() > SIZE_MAX) {
        yawSeries.getData().remove(0);
    }
}
}

```

Listing 12-8 shows the code of `OrientationFX`, the `Application` class, where you'll add a `BorderPane` to include a pair of buttons on top, to start and stop the serial communication, and the `ArduinoChart` instance on the center.

Listing 12-8. The `OrientationFX` Class

```

/**
 * @author Jose Pereda
 */
public class OrientationFX extends Application {

    private static final double OFFSET_YAW = -180;
    private Serial serial;
    private ArduinoChart arduinoChart;

    private TextField textYaw;

    private final BooleanProperty connection = new SimpleBooleanProperty();

```

```

private final ChangeListener<String> listener = (obs, ov, nv) -> {
    String[] split = nv.split("\\s");
    long time = Long.parseLong(split[0]);
    double offset = OFFSET_YAW;
    if (!textYaw.getText().isEmpty()) {
        try {
            offset = Double.parseDouble(textYaw.getText());
        } catch (NumberFormatException nfe) { }
    }
    double yaw = Double.parseDouble(split[1]) + offset;
    double pitch = - Double.parseDouble(split[2]);
    double roll = - Double.parseDouble(split[3]);
    Platform.runLater(() -> {
        arduinoChart.processEvent(time, roll, pitch, yaw);
    });
};

@Override
public void start(Stage primaryStage) throws Exception {
    arduinoChart = new ArduinoChart();

    BorderPane root = new BorderPane();

    Button buttonStart = new Button();
    buttonStart.setText("Start");
    buttonStart.setOnAction(e -> startSerial());

    Button buttonStop = new Button();
    buttonStop.setText("Stop");
    buttonStop.setOnAction(e -> stopSerial());

    Label labelConnection = new Label("Not connected");
    connection.addListener((obs, ov, nv) -> labelConnection.setText(nv ?
        "Connected to: " + serial.getPortName() : "Not connected"));
    labelConnection.setPrefWidth(400);

    textYaw = new TextField("") + OFFSET_YAW;
    HBox top = new HBox(buttonStart, buttonStop, new Label("Yaw Offset (°): "), textYaw);
    top.getStyleClass().add("box");
    root.setTop(top);

    final VBox chartBox = arduinoChart.createArduinoChart();
    chartBox.setPrefWidth(500);
    HBox center = new HBox(chartBox);
    center.getStyleClass().add("box");
    center.setPrefHeight(500);
    root.setCenter(center);
}

```

```

HBox bottom = new HBox(labelConnection);
bottom.getStyleClass().add("box");
root.setBottom(bottom);

Scene scene = new Scene(root, 1000, 600);
scene.getStylesheets().add(OrientationFX
    .class.getResource("arduino.css").toExternalForm());

primaryStage.setTitle("OrientationFX");
primaryStage.setScene(scene);
primaryStage.show();
}

@Override
public void stop(){
    stopSerial();
}

private void startSerial(){
    if (serial == null) {
        serial = new Serial();
    }
    serial.getLine().addListener(listener);
    serial.connect();
    connection.set(!serial.getPortName().isEmpty());
}

private void stopSerial(){
    if (serial != null) {
        serial.disconnect();
        serial.getLine().removeListener(listener);
        connection.set(false);
        serial = null;
    }
}
}

```

Listing 12-9 shows the CSS file added as a style sheet for the application.

Listing 12-9. The arduino.css File

```

.root {
    -fx-background-color: derive(beige, 60%);
    -fx-font: 16px "Tahoma";
}

.box {
    -fx-padding: 10px;
    -fx-spacing: 20px;
    -fx-alignment: center-left;
}

```

```
.chart {  
    -fx-padding: 0px;  
}  
  
.chart-plot-background {  
    -fx-background-color: linear-gradient(to bottom, derive(beige, 40%), derive(beige, 20%));  
}  
  
.axis:bottom {  
    -fx-border-color: derive(beige, -20%) transparent transparent transparent;  
}  
  
.axis:left {  
    -fx-border-color: transparent derive(beige, -20%) transparent transparent;  
}  
  
.axis-tick-mark,  
.axis-minor-tick-mark,  
.chart-vertical-grid-lines,  
.chart-horizontal-grid-lines,  
.chart-vertical-zero-line,  
.chart-horizontal-zero-line {  
    -fx-stroke: derive(beige, -20%);  
}  
  
.chart-legend {  
    -fx-background-color: linear-gradient(to bottom, derive(beige, 40%), derive(beige, 20%));  
    -fx-padding: 10px;  
}  
  
.chart-series-line {  
    -fx-stroke-width: 3px;  
}  
  
.default-color0.chart-series-line { -fx-stroke: blueviolet; }  
.default-color1.chart-series-line { -fx-stroke: orangered; }  
.default-color2.chart-series-line { -fx-stroke: firebrick; }  
  
.default-color0.chart-line-symbol {  
    -fx-background-color: transparent, blueviolet;  
    -fx-background-radius: 3px;  
}  
  
.default-color1.chart-line-symbol {  
    -fx-background-color: transparent, orangered;  
    -fx-background-radius: 3px;  
}  
.default-color2.chart-line-symbol {  
    -fx-background-color: transparent, firebrick;  
    -fx-background-radius: 3px;  
}
```

Building and Running the Project

To build and run the project from NetBeans, click on Run Project. Check that you have set `com.jfxbe.orientationfx.OrientationFX` as `mainClass` (choose Properties ► Run).

To build the module and run the project from the command line with Java 9, run the steps from Listing 12-10. Be aware that in order to build the module, you need to include the `jssc.jar` file with the `module-path` option.

Listing 12-10. Running the OrientationFX Project

```
javac --module-path libs/jssc.jar -d mods/com.jfxbe.orientationfx
    $(find src/com.jfxbe.orientationfx -name "*.java")
cp src/com.jfxbe.orientationfx/classes/com/jfxbe/orientationfx/arduino.css
    mods/com.jfxbe.orientationfx/com/jfxbe/orientationfx
java --module-path mods:libs -m com.jfxbe.orientationfx/com.jfxbe.orientationfx.
OrientationFX
```

If everything is in place, and you have your Arduino 101 plugged in, you should be able to hold it and wave it and get something like the Figure 12-22.

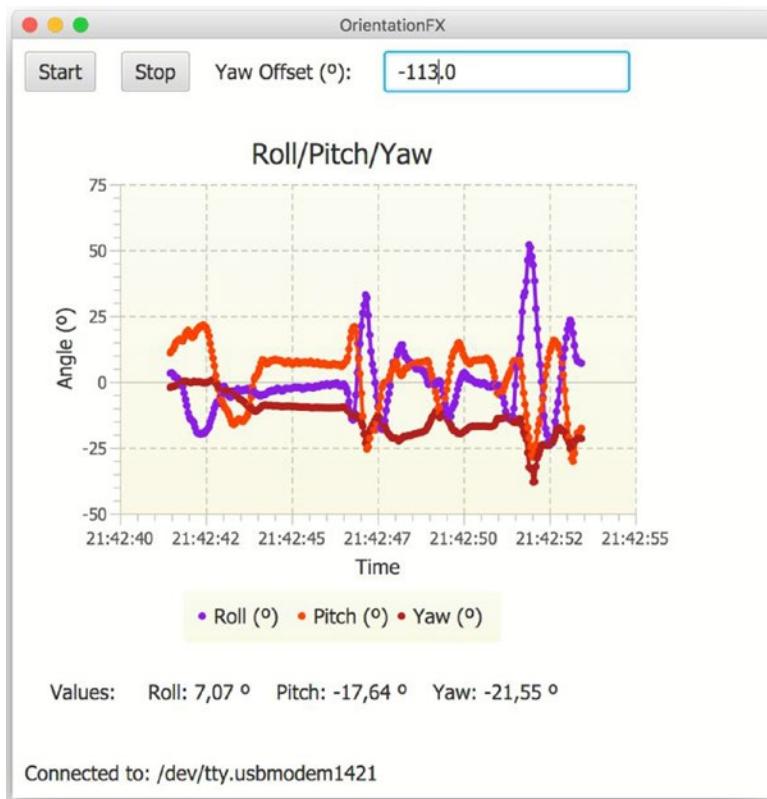


Figure 12-22. Running the OrientationFX example

How It Works

When the application starts, it creates an instance of `ArduinoChart`.

```
@Override
public void start(Stage primaryStage) throws Exception {
    arduinoChart = new ArduinoChart();
```

Inside the `arduinoChart` class is a `LineChart`, with three series that take pairs of numbers to be plotted against each axis, which are `NumberAxis` instances. Several limits are defined for these series: the maximum number of points in each of them will be 300:

```
private static final int SIZE_MAX = 300;

private LineChart<Number,Number> chart;
private Series<Number, Number> rollSeries, pitchSeries, yawSeries;
private NumberAxis xAxis, yAxis;
```

Then you create the chart, starting by creating both axes. For the `xAxis` to show formatted data, this example overrides the `toString()` method, so every long value in milliseconds will be written in `HH:mm:ss` format:

```
public VBox createArduinoChart() {
    xAxis = new NumberAxis();
    xAxis.setLabel("Time");
    xAxis.setAutoRanging(true);
    xAxis.setForceZeroInRange(false);
    xAxis.setTickLabelFormatter(new StringConverter<Number>() {
        @Override
        public String toString(Number t) {
            return new SimpleDateFormat("HH:mm:ss")
                .format(new Date(t.longValue()));
        }
        @Override
        public Number fromString(String string) {
            throw new UnsupportedOperationException("Not supported yet.");
        }
    });
    yAxis = new NumberAxis();
    yAxis.setLabel("Angle (°)");
```

Then it creates the chart by specifying its axes. It sets a few properties: `setCreateSymbols(true)` will plot a big white circle on every point of the series. You can adjust the default size and color with CSS. `setAnimated(false)` avoids the animation of the chart after adding any new point. Because you are plotting points at a high frequency, it's better this way. For lower frequencies, on the other hand, you will set it to `true` to create a smooth transition effect. `setLegendVisible(true)` shows the legend with the series names:

```
chart = new LineChart<>(xAxis, yAxis);
chart.setCreateSymbols(true);
chart.setAnimated(false);
chart.setLegendVisible(true);
chart.setTitle("Roll/Pitch/Yaw");
```

You then create the three series, set the names for them (which will be displayed in the legend box), and add them to the Data chart, which is an `ObservableList<Series>`:

```
rollSeries = new Series<>();
rollSeries.setName("Roll (°)");
pitchSeries = new Series<>();
pitchSeries.setName("Pitch (°)");
yawSeries = new Series<>();
yawSeries.setName("Yaw (°)");
chart.getData().addAll(rollSeries, pitchSeries, yawSeries);
```

Finally, the chart is added to a `VBox`, along with an `HBox` with the labels for the instant values:

```
labelRoll = new Label();
labelPitch = new Label();
labelYaw = new Label();

final HBox hBox = new HBox(new Label("Values: "), labelRoll, labelPitch, labelYaw);
hBox.getStyleClass().add("box");

final VBox vBox = new VBox(chart, hBox);
vBox.getStyleClass().add("box");
return vBox;
}
```

Back to the application class, a listener will be created to bind the presence of any new line on the serial port (corresponding to a reading of the orientation angles coming from the Arduino 101) with the action of adding these values as new pairs to the series. For that, you set the x coordinate value as the time when you add the reading in milliseconds (on the chart it will be formatted to HH:mm:ss) and the y coordinate value is a double measurement of any of the angles reported by the Arduino in the String `nv`. You parse the strings to double. For the yaw angle, you allow adding an offset value with a `TextField`, given that, after calibration, this angle is not always reset to 0°. Also, notice the minus sign for pitch and roll: It is due to the change of the coordinate system between Arduino and JavaFX.

```
private final ChangeListener<String> listener = (obs, ov, nv) -> {
    String[] split = nv.split("\\s");
    long time = Long.parseLong(split[0]);
    double offset = OFFSET_YAW;
    if (!textYaw.getText().isEmpty()) {
        try {
            offset = Double.parseDouble(textYaw.getText());
        } catch (NumberFormatException nfe) { }
    }
    double yaw = Double.parseDouble(split[1]) + offset;
    double pitch = - Double.parseDouble(split[2]);
    double roll = - Double.parseDouble(split[3]);
```

By using `Platform.runLater()`, you can pass the values to `ArduinoChart`, to render them in the chart. This is mandatory, as the Serial task is running outside the JavaFX thread. With `Platform.runLater()`, not only do you place the task of filling the series with the incoming data in the JavaFX thread, but also you give to the Scene graph the required time to render the chart. Some of the angles might not be processed if the rate is too high.

```
Platform.runLater(() -> {
    arduinoChart.processEvent(time, roll, pitch, yaw);
});
```

As the series data could grow indefinitely, to avoid memory issues, you set a `SIZE_MAX` value of 300 points and remove the first values of the series if you exceed that limit.

```
public void processEvent(long time, double roll, double pitch, double yaw) {
    rollSeries.getData().add(new Data(time, roll));
    labelRoll.setText("Roll: " + String.format("%3.2f °", roll));
    if (rollSeries.getData().size() > SIZE_MAX) {
        rollSeries.getData().remove(0);
    }

    pitchSeries.getData().add(new Data(time, pitch));
    labelPitch.setText("Pitch: " + String.format("%3.2f °", pitch));
    if (pitchSeries.getData().size() > SIZE_MAX) {
        pitchSeries.getData().remove(0);
    }

    yawSeries.getData().add(new Data(time, yaw));
    labelYaw.setText("Yaw: " + String.format("%3.2f °", yaw));
    if (yawSeries.getData().size() > SIZE_MAX) {
        yawSeries.getData().remove(0);
    }
}
```

Once you have the chart ready, you create the scene, add it to the stage, and finally display it. The example creates a `BorderPane`, so in the center you add the chart, and on the bottom you add a `Label` to show the connection status including the port name. Note the styling done to the root pane.

```
BorderPane root = new BorderPane();

final VBox chartBox = arduinoChart.createArduinoChart();
chartBox.setPrefWidth(500);
HBox center = new HBox(chartBox); //, arduino3D.createArduino3D());
center.getStyleClass().add("box");
center.setPrefHeight(500);
root.setCenter(center);
```

```
Scene scene = new Scene(root, 600, 600);
scene.getStylesheets().add(OrientationFX
    .class.getResource("arduino.css").toExternalForm());

primaryStage.setTitle("OrientationFX");
primaryStage.setScene(scene);
primaryStage.show();
```

Finally, you add two buttons to start and stop the serial task, and the `textfield` to modify the yaw offset:

```
Button buttonStart = new Button();
buttonStart.setText("Start");
buttonStart.setOnAction(e -> startSerial());

Button buttonStop = new Button();
buttonStop.setText("Stop");
buttonStop.setOnAction(e -> stopSerial());
textYaw = new TextField("") + OFFSET_YAW;
HBox top = new HBox(buttonStart, buttonStop, new Label("Yaw Offset (°): "), textYaw);
top.getStyleClass().add("box");
root.setTop(top);
```

In the `startSerial` method, you can try to open the serial port and start reading from it. The listener will be added to `serial.getLine()` and connection will be set to true in that case:

```
private void startSerial(){
    if (serial == null) {
        serial = new Serial();
    }
    serial.getLine().addListener(listener);
    serial.connect();
    connection.set(!serial.getPortName().isEmpty());
}
```

Whenever the application is closed, you must remove the `listener` and call the `serial.disconnect()` method, to shut down the serial connection properly (see Listing 12-5).

```
private void stopSerial(){
    if (serial != null) {
        serial.disconnect();
        serial.getLine().removeListener(listener);
        connection.set(false);
        serial = null;
    }
}
```

Adding More Functionality

If you have come this far, and the example is working for you, you will want to continue upgrading the application with a new extra feature. It would be nice if you could somehow render a JavaFX 3D Arduino board and rotate it based on the orientation angles coming from the real Arduino.

Listing 12-11 contains the source code of the `Arduino3D` class, which simulates a very simple 3D board with a few JavaFX 3D Box nodes.

Listing 12-11. The `Arduino3D` Class

```
public class Arduino3D {

    private static final int SIZE = 5;

    private Affine affine;

    private final List<Double> averageRoll = new ArrayList();
    private final List<Double> averagePitch = new ArrayList();
    private final List<Double> averageYaw = new ArrayList();

    public VBox createArduino3D() {
        Box board = new Box(300, 10, 200);
        board.setMaterial(new PhongMaterial(Color.web("#008282")));
        Box longHeader = new Box(210, 40, 10);
        longHeader.setMaterial(new PhongMaterial(Color.web("#505050")));
        longHeader.getTransforms().add(new Translate(40, -20, 90));
        Box shortHeader = new Box(170, 40, 10);
        shortHeader.setMaterial(new PhongMaterial(Color.web("#505050")));
        shortHeader.getTransforms().add(new Translate(60, -20, -90));

        Group arduino = new Group(board, longHeader, shortHeader);
        affine = new Affine();
        arduino.getTransforms().add(affine);

        SubScene subScene = new SubScene(new Group(arduino), 400, 400,
            true, SceneAntialiasing.BALANCED);
        PerspectiveCamera camera = new PerspectiveCamera();
        camera.setTranslateX(-200);
        camera.setTranslateY(-200);
        camera.setTranslateZ(-50);
        subScene.setCamera(camera);

        VBox vBox = new VBox(new Label("3D Rendering"), subScene);
        vBox.getStyleClass().add("box");
        return vBox;
    }

    public void processEvent(double roll, double pitch, double yaw) {
        double avYaw = average(averageYaw, Math.toRadians(yaw));
        double avPitch = average(averagePitch, Math.toRadians(pitch));
        double avRoll = average(averageRoll, Math.toRadians(roll));
        matrixRotateNode(avRoll, avPitch, avYaw);
    }
}
```

```

private void matrixRotateNode(double roll, double pitch, double yaw) {
    double mxx = Math.cos(pitch) * Math.cos(yaw);
    double mxy = Math.cos(roll) * Math.sin(pitch) +
        Math.cos(pitch) * Math.sin(roll) * Math.sin(yaw);
    double mxz = Math.sin(pitch) * Math.sin(roll) -
        Math.cos(pitch) * Math.cos(roll) * Math.sin(yaw);
    double myx = -Math.cos(yaw) * Math.sin(pitch);
    double myy = Math.cos(pitch) * Math.cos(roll) -
        Math.sin(pitch) * Math.sin(roll) * Math.sin(yaw);
    double myz = Math.cos(pitch) * Math.sin(roll) +
        Math.cos(roll) * Math.sin(pitch) * Math.sin(yaw);
    double mzx = Math.sin(yaw);
    double mzy = -Math.cos(yaw) * Math.sin(roll);
    double mzz = Math.cos(roll) * Math.cos(yaw);

    affine.setToTransform(mxx, mxy, mxz, 0,
        myx, myy, myz, 0,
        mzx, mzy, mzz, 0);
}

private double average(List<Double> list, double value) {
    while (list.size() > SIZE) {
        list.remove(0);
    }
    list.add(value);

    return list.stream()
        .collect(Collectors.averagingDouble(d -> d));
}
}

```

Listing 12-12 contains the modifications you need to do to the OrientationFX class to include an instance of the Arduino3D class and render the sub-scene, and also to send the events from the serial task and rotate the 3D board.

Listing 12-12. A Snippet of the OrientationFX Class

```

public class OrientationFX extends Application {

    private Arduino3D arduino3D;

    private final ChangeListener<String> listener = (obs, ov, nv) -> {

        Platform.runLater(() -> {
            arduinoChart.processEvent(time, roll, pitch, yaw);
            arduino3D.processEvent(roll, pitch, yaw);
        });
    };
}

```

```

@Override
public void start(Stage primaryStage) throws Exception {
    arduinoChart = new ArduinoChart();
    arduino3D = new Arduino3D();

    BorderPane root = new BorderPane();
    HBox center = new HBox(chartBox, arduino3D.createArduino3D());
    root.setCenter(center);

    Scene scene = new Scene(root, 600, 600);
    primaryStage.setScene(scene);
    primaryStage.show();
}
}
}

```

Building and Running the Project

To build and run the project from NetBeans, just click on Run Project. To build the module and run the project from the command line with Java 9, run the steps mentioned in Listing 12-10.

Plug your Arduino 101 in, click the Start button, and hold and wave the Arduino. You should get something like Figure 12-23, where the 3D board orientation mimics the real one with enough precision.

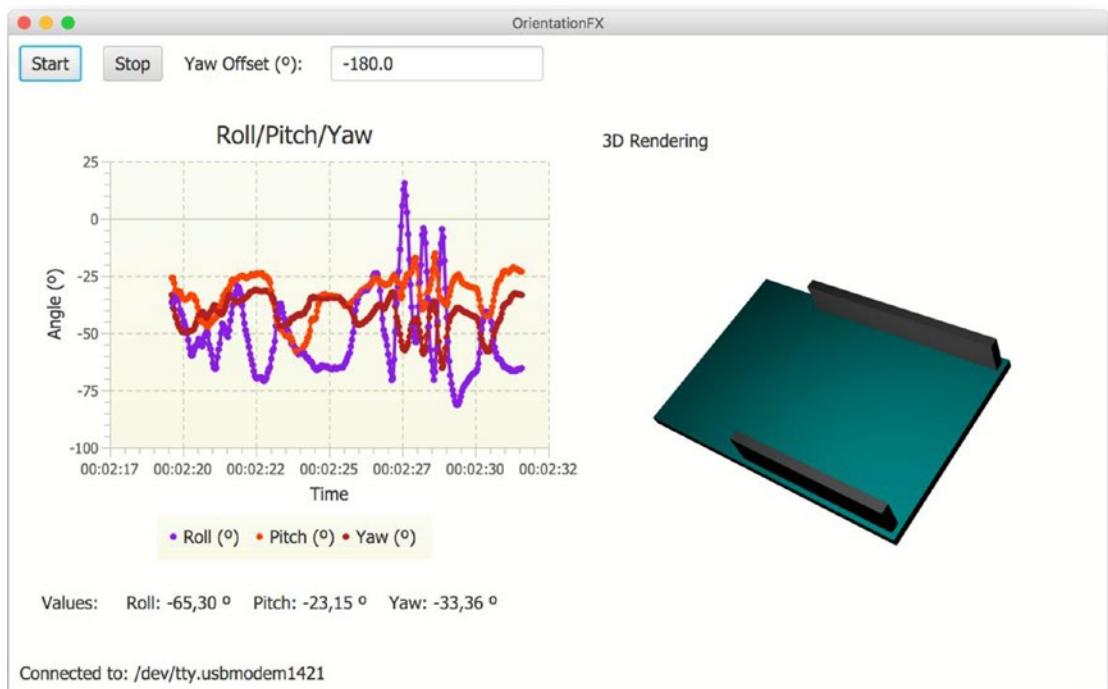


Figure 12-23. Running the advanced OrientationFX example

Note that the 0,0,0 position corresponds with the board in your hands in such a way that you can read the Genuino 101 logo from left to right, and the power plug stays to the left. The yaw value can be adjusted if required.

How It Works

The example creates an instance of `Arduino3D` by adding three boxes to a `Group`:

```
public VBox createArduino3D() {
    Box board = new Box(300, 10, 200);
    board.setMaterial(new PhongMaterial(Color.web("#008282")));
    Box longHeader = new Box(210, 40, 10);
    longHeader.setMaterial(new PhongMaterial(Color.web("#505050")));
    longHeader.getTransforms().add(new Translate(40, -20, 90));
    Box shortHeader = new Box(170, 40, 10);
    shortHeader.setMaterial(new PhongMaterial(Color.web("#505050")));
    shortHeader.getTransforms().add(new Translate(60, -20, -90));
    Group arduino = new Group(board, longHeader, shortHeader);
```

Then it adds this group to a `SubScene` inside another group. The sub-scene allows you to use a camera without affecting the rest of the scene:

```
SubScene subScene = new SubScene(new Group(arduino), 400, 400,
    true, SceneAntialiasing.BALANCED);
PerspectiveCamera camera = new PerspectiveCamera();
camera.setTranslateX(-200);
camera.setTranslateY(-200);
camera.setTranslateZ(-50);
subScene.setCamera(camera);

VBox vBox = new VBox(new Label("Rendering 3D"), subScene);
vBox.getStyleClass().add("box");
return vBox;
```

The `Arduino` group adds an *affine* to its list of transforms. You add this group to a `SubScene` inside another group. The sub-scene allows you to use a camera without affecting the rest of the scene:

```
affine = new Affine();
arduino.getTransforms().add(affine);
```

So now, whenever there is a serial event and a new set of orientation angles is provided, all you need to do is update the affine coefficients. For that, you have to combine the three rotations into a single rotation matrix:

```
public void processEvent(double roll, double pitch, double yaw) {
    matrixRotateNode(roll, pitch, yaw);
}
```

```

private void matrixRotateNode(double roll, double pitch, double yaw) {
    double mxx = Math.cos(pitch) * Math.cos(yaw);
    double mxy = Math.cos(roll) * Math.sin(pitch) +
        Math.cos(pitch) * Math.sin(roll) * Math.sin(yaw);
    double mxz = Math.sin(pitch) * Math.sin(roll) -
        Math.cos(pitch) * Math.cos(roll) * Math.sin(yaw);
    double myx = -Math.cos(yaw) * Math.sin(pitch);
    double myy = Math.cos(pitch) * Math.cos(roll) -
        Math.sin(pitch) * Math.sin(roll) * Math.sin(yaw);
    double myz = Math.cos(pitch) * Math.sin(roll) +
        Math.cos(roll) * Math.sin(pitch) * Math.sin(yaw);
    double mzx = Math.sin(yaw);
    double mzy = -Math.cos(yaw) * Math.sin(roll);
    double mzz = Math.cos(roll) * Math.cos(yaw);

    affine.setToTransform(mxx, mxy, mxz, 0,
        myx, myy, myz, 0,
        mzx, mzy, mzz, 0);
}

```

The math involved in this process is out of scope for now, but a good reference can be found at https://en.wikipedia.org/wiki/Rotation_formalisms_in_three_dimensions.

The previous code will give an instant rotation for every set of angles, so in order to make it a little bit steadier, the use of the average of the last five values for each angle is more convenient.

```

private static final int SIZE = 5;
private final List<Double> averageRoll = new ArrayList();
private final List<Double> averagePitch = new ArrayList();
private final List<Double> averageYaw = new ArrayList();

public void processEvent(double roll, double pitch, double yaw) {
    double avYaw = average(averageYaw, Math.toRadians(yaw));
    double avPitch = average(averagePitch, Math.toRadians(pitch));
    double avRoll = average(averageRoll, Math.toRadians(roll));
    matrixRotateNode(avRoll, avPitch, avYaw);
}

private double average(List<Double> list, double value) {
    while (list.size() > SIZE) {
        list.remove(0);
    }
    list.add(value);

    return list.stream()
        .collect(Collectors.averagingDouble(d -> d));
}

```

More Examples

There are plenty of resources out there to demonstrate or utilize the Arduino, and you can experiment whatever you have in mind! Arduino and the required equipment are inexpensive and safe (in most cases), and they give you the chance to improve your do-it-yourself (DIY) skills.

Look for an affordable project buy what you need, build it, test it first on the Arduino IDE, and then design a JavaFX application to monitor and control your devices.

Share your developments and look for help when you need it; there are great communities of people willing to help, in both Arduino development and JavaFX.

Summary

In this chapter you've discovered what can be done by combining Arduino and JavaFX. You began by learning about Arduino, its different boards, and main specifications. Next, you saw how to communicate with it from the Arduino IDE. After learning about loading examples in the IDE, you got the chance to try it for yourself an Arduino 101, uploading a sketch and reading the orientation angles in the Serial Monitor.

Then you learned about the different ways to read a serial port from your computer and found about the Java Simple Serial Connector library. Installing this third-party component allowed you to perform serial port operations such as locating, opening, reading, writing, and closing.

To monitor and display the Arduino readings, you learned how to use the JavaFX Charts API. Then you saw an example in which a JavaFX chart is plotted with a series of data coming from the serial port, using an Arduino board that comes with an six-axis accelerometer and gyroscope, to provide the three orientation angles. Finally, you discovered how to enhance the application by adding a JavaFX 3D simulation of the real board and rotate it based on the serial readings.

CHAPTER 13



JavaFX on Mobile

We all have heard many times the WORA slogan (“write once, run anywhere”), and while it is true that Java runs anywhere you have a JVM, that’s not the case on modern mobile devices. While the latest trends confirm that smartphones are taking over computers, why can’t we target that huge market yet? Luckily, the open source project JavaFXPorts was born to allow all Java developers to deploy their projects on mobile (Android and iOS).

In this chapter, you learn about the JavaFXPorts project, and how this led to Gluon Mobile, a library created to enhance the mobile experience of the JavaFX desktop design.

As a full example, in this chapter, you will see how to create a simple JavaFX application with Gluon Mobile: `BasketStats`. It is a simple app to track the annotation during a basketball game, and you will deploy it to Android and iOS.

Before getting into the example, let’s get started with JavaFXPorts and Gluon Mobile.

JavaFXPorts: The Port to Mobile

JavaFXPorts is the open source project that brings Java and JavaFX to mobile and embedded hardware. This includes iPhone, iPad, Android devices, and even the Raspberry Pi. The different projects that make possible this *miracle* can be found here: <https://bitbucket.org/javafxports>.

This adventure started at the end of 2013, when Johan Vos and his team started the work on porting JavaFX to Android, based on the OpenJFX project. In parallel, RoboVM started as a similar open source project to port Java to iOS. In February 2015, LodgOn and Trillian Mobile, the companies behind those projects, announced a joint effort to combine the best of both projects into a single plug-in, called **jfxmobile-plugin**. The one and only Gradle JavaFX plug-in for mobile was created and freely available through the JavaFXPorts repository.

A few weeks later, the Gluon company (see <http://gluonhq.com>) was created to gather all the efforts around the JavaFXPorts project and to deliver Gluon Mobile, a lightweight application framework and a set of mobile controls to enhance the mobile experience.

JavaFXPorts Under the Hood

When we create and compile a Java application, the result is Java bytecode. To run it, you need the JRE, containing native libraries specific for each platform. Since there is no JRE for mobile devices, a different approach is required.

On Android, the Google’s Android SDK contains tools for bundling applications, resources, and libraries into an Android package (APK). The `jfxmobile` plug-in will use these tools to create and install the Java mobile app, on top of the Android’s Dalvik/Art VMs, relatively similar to the Java VM.

On iOS, the JRE needs to be packaged inside the app. The RoboVM ahead-of-time compiler is used to translate the Java code into native iOS code, linking the required runtime libraries with the application.

Both the Android runtime and the iOS AOT compiler currently use the Apache Harmony implementation of the Java class libraries, which is only a partial implementation of Java 7, and the project is officially abandoned. Adding to this, the RoboVM project has been stopped after Microsoft acquired Xamarin, which acquired RoboVM in the first place.

At this moment, this means that JavaFXPorts supports most of the Java 7 SE APIs, and a few Java 8 APIs are supported as well (like lambda expressions, but not streams).

Because of these limitations, Gluon is working on GluonVM, a high-performance Java 8/9 VM that will leverage the OpenJDK class libraries and provide full Java 9 functionality on mobile devices. It is expected to be available at the time Java 9 is released.

Note Until GluonVM is released and given the mentioned limitations, to run JavaFX on Mobile we can't use Java 9 nor the modular system. That is why the examples in this chapter use the existing implementation (Java 7 and some features of Java 8).

Getting Started with JavaFXPorts

The documentation to get started creating Java mobile applications with JavaFXPorts can be found here: <http://docs.gluonhq.com/javafxports/>.

The Gradle plug-in jfxmobile-plugin does almost all the required work. By including this plug-in in your regular JavaFX application, it will automatically create a number of tasks for you, and it will package your application as a native iOS or a native Android package.

These are the prerequisites before starting a Java mobile project. On your machine:

- Install the latest JDK 8 version for your development machine. Get it from here: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.
- Install Gradle 2.2 or higher from <https://gradle.org/install>. It is required to build applications with the jfxmobile plug-in.

If you want to deploy on Android:

- Install the Android SDK from <https://developer.android.com/studio/index.html>. You can download Android Studio, which bundles the SDK and the required Android tools, or you can just download and install the SDK tools at the end of the link.
- Run the Android SDK Manager (Android Studio -> Tools -> Android -> SDK Manager or the command line <android sdk>/tools/android) and then install at least version 23.0.1 of Build-tools, the SDK Platform for APIs 21 up to 25, from Extras the Android Support Repository.
- Create a properties file in <Users>/<User>/.gradle/gradle.properties and add the ANDROID_HOME=<path.to.Android.sdk> property.

Before deploying to an Android device, you need to follow these steps on the device:

- Go to Settings -> About phone -> Build Number and tap seven times on it to enable developer mode.
- Go to Settings -> Development Options -> USB Debugging and select enable.
- Go to Settings -> Security -> Unknown sources and enable installing apps from unknown sources.

If you want to deploy on iOS:

- You need a Mac with MacOS X 10.11.5 or higher
- You need Xcode 8.x or higher, available from the Mac App Store

Before deploying to an iOS device, you need to get a provisioning profile from the Apple Developer portal. See this link on how to get a free one to deploy to your own device: http://docs.gluonhq.com/javafxports/#_ios_3.

Hello Mobile World Example

With all the prerequisites in place, let's create a Hello Mobile World example that you can run on desktop and mobile platforms.

On your favorite IDE, you can create a new Gradle project. Give it a name (`HelloMobile`), a location, and a main class, such as `org.jfxbe.chap13.HelloMobile`.

Edit the `build.gradle` file and add the content from Listing 13-1. It will apply the `jfxmobile` plug-in to the project. Save and reload the project to update the project. The first time this is done, the plug-in will download and install some internal dependencies, so it may take a while.

Listing 13-1. The `build.gradle` File

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'org.javafxports:jfxmobile-plugin:1.3.5'
    }
}

apply plugin: 'org.javafxports.jfxmobile'

repositories {
    jcenter()
}

mainClassName = 'org.jfxbe.chap13.HelloMobile'

jfxmobile {
    android {
    }
    ios {
        forceLinkClasses = [ 'org.jfxbe.chap13.**.*' ]
    }
}
```

Note that the version of the plug-in at the time of this writing was 1.3.5. To keep it updated, check out <https://bitbucket.org/javafxports/javafxmobile-plugin>.

Now in the main class, extend Application and create a JavaFX Scene with some content, as shown in Listing 13-2.

Listing 13-2. Hello Mobile Main Class

```
package org.jfxbe.chap13;

import javafx.application.Application;
import javafx.geometry.Rectangle2D;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Screen;
import javafx.stage.Stage;

public class HelloMobile extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        Rectangle2D bounds = Screen.getPrimary().getVisualBounds();
        final Button button = new Button("Hello Mobile World!");
        button.setStyle("-fx-font-size: 20pt;");
        button.setOnAction(e -> button.setRotate(button.getRotate() - 30));

        final StackPane stackPane = new StackPane(button);

        Scene scene = new Scene(stackPane, bounds.getWidth(), bounds.getHeight());
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Finally, you can build the application and run it on your platforms. To run a Gradle build, just type gradle followed by the desired task name, e.g. gradle build will build the project. Run gradle run to run it on your developer machine, gradle androidInstall to deploy it on an Android device, or gradle launchIOSDevice to an iOS device. Figure 13-1 shows a screenshot of the app running on Android.



Figure 13-1. Hello Mobile World deployed on Android

How Does It Work?

The jfxmobile plug-in adds a number of tasks to your Java application that allow you to create packages that can be uploaded to the Apple App Store and the Android Play Store.

The jfxmobile plug-in downloads and installs the required JavaFX SDKs:

- Dalvik SDK for Android: The plug-in currently depends on org.javafxports:dalvik-sdk:8.60.9 and org.javafxports:jfxdvk:8.60.9.
- Retrolambda plug-in: Transforms the code to Java 6 compatible bytecode and currently depends on net.orfjackal.retrolambda:retrolambda:2.5.1.
- iOS SDK: The plug-in currently depends on org.javafxports:ios-sdk:8.60.9.
- MobiDevelop: The fork of RoboVM 1.8.0 that's used to compile the code to iOS compatible bytecode. Currently depends on com.mobidevelop.robovm:robovm-dist:tar.gz:nocompiler:2.3.0.

The source code for the mentioned SDKs is available here: <http://bitbucket.org/javafxports/8u-dev-rt>.

On Android, the plug-in will then execute a number of commands that will lead to the creation of an Android package (APK) in the directory build/javafxports/android. The package will be deployed and installed in the device. Listing 13-3 shows the output of a successful deploy in Android, where all the involved tasks are listed.

Note When running Gradle tasks, use `--info`, `--debug`, or `--stacktrace` to get a more verbose output that may help you pin pointing errors and issues, such as `gradle --info androidInstall`.

Listing 13-3. Output When Running `androidInstall`

```
$ gradle androidInstall
:validateManifest
:collectMultiDexComponents
:compileJava
:processResources UP-TO-DATE
:classes
:compileAndroidJava UP-TO-DATE
:copyClassesForRetrolambda
:applyRetrolambda
Retrolambda 2.5.1
:mergeClassesIntoJar
:shrinkMultiDexComponents
:createMainDexList
:writeInputListFile
:dex
:mergeAndroidAssets
:mergeAndroidResources
:processAndroidResources UP-TO-DATE
:processAndroidResourcesDebug
:validateSigningDebug
:apkDebug
:zipalignDebug
:androidInstall
Installed on device.
```

BUILD SUCCESSFUL

Total time: 41.168 secs

On iOS, the situation is internally different, but the Gradle commands are similar. The plug-in will download and install the RoboVM compiler, and it will use RoboVM compiler commands to create an iOS application in `build/javafxports/ios`.

Note When running the app on your mobile device, the standard output can be accessed on Android devices by running `<path.to.android skd>/platform-tools/adb logcat`. On the Mac, open Xcode ➤ Windows ➤ Devices on iOS to access the output.

Finally, Gradle uses convention over configuration, and the default configuration applied to the plug-in makes the `build.gradle` file extremely simple. You can modify this configuration when required. Check the documentation at <http://docs.gluonhq.com/javafxports/#anchor-3> for a full list of modifiable properties for Android and iOS.

Submitting the App to the Stores

When you have finished testing the app on your mobile devices, you can proceed to upload it to the Apple App Store and the Android Play Store. To do so, the plug-in already includes the required tasks.

On Android, you need to provide valid icon images (under `/src/android/res`) and you have to disable the debugging options in the `AndroidManifest` file: Under the `application` tag, add `android:debuggable="false"`. And finally you have to sign the app, as explained at <https://developer.android.com/studio/publish/app-signing.html>. You can add the signing configuration to your build `gradle` file, as shown in Listing 13-4.

Listing 13-4. Signing Configuration on Android

```
jfxmobile {
    android {
        signingConfig {
            storeFile file("path/to/my-release-key.keystore")
            storePassword 'STORE_PASSWORD'
            keyAlias 'KEY_ALIAS'
            keyPassword 'KEY_PASSWORD'
        }
        manifest = 'lib/android/AndroidManifest.xml'
    }
}
```

If everything is set, run `gradle androidRelease` to generate the APK under `/build/javafxports/android`. To upload it to the store, you'll have to enroll with Google Play Developers (<https://play.google.com/apps/publish/signup/>) and fill in the required forms.

On iOS, provide the icons for the app, and once you have enrolled to the Apple Developer program, you need to sign the app with a provisioning profile for production, as shown in Listing 13-5. Run `gradle createIpa` and submit the app at `/build/javafxports/ios` through Xcode ➤ Open Developer Tool ➤ Application Loader. Fixing the initial requirements that the tool reveals as missing can be done by adding the proper keys to the `plist` file under `/src/ios/Default-Info.plist`. When this is done, go to iTunes Connect (<https://itunesconnect.apple.com>) to prepare the app for release. As with any other regular iOS app, it usually takes a few iterations to get the app approved for submission, until you fulfill all the requirements from the reviewers.

Listing 13-5. Signing Configuration on iOS

```
jfxmobile {
    ios {
        arch = "arm64"
        infoPList = file('src/ios/Default-Info.plist')
        forceLinkClasses = ['your.package.*.*',...]
        iosProvisioningProfile = 'MyApp'
        iosSignIdentity = 'iPhone Distribution: *****'
    }
}
```

Gluon Mobile

The Gluon Mobile library is designed to help developers create high performance, great looking, and cloud connected mobile apps from a single Java codebase, for iOS and Android. It improves time-to-market for applications, by providing APIs for modern material design user interfaces, accessing mobile device features, connecting to web services, and synchronizing state across many devices.

Gluon Mobile is the client-side library and development tool that:

- Provides UI controls for the client application, using Glisten, the UI component that provides a native look and feel, with JavaFX controls and specific layouts.
- Handles communication with the server-side Gluon CloudLink, using Connect, an open source project that allows communications with the Gluon CloudLink itself and other web services. Find out more at <http://gluonhq.com/products/mobile/connect/>.
- Abstracts (parts of) the platform-specific APIs by using *Charm Down*, an open source project that offers a collection of services like filesystem, local or push notifications, GPS, sensors, camera,... More details can be found at <http://gluonhq.com/products/mobile/charm-down/>.

To get started using Gluon Mobile, you can create a Gradle project from the scratch, modify any an existing samples (<http://gluonhq.com/support/samples/>), or install and use the Gluon plug-in for your IDE.

The Gluon IDE Plug-Ins

The Gluon plug-in for your IDE aids in creating a Gluon application project inside your IDE: there are plug-ins for:

- NetBeans. See instructions: <http://docs.gluonhq.com/charm/latest/#netbeans-plugin>.
- IntelliJ IDEA. Follow: <http://docs.gluonhq.com/charm/latest/#intellij-plugin>.
- Eclipse. See: <http://docs.gluonhq.com/charm/latest/#eclipse-plugin>.

Once you have installed the plug-in, select New Project, go to the Gluon category, and pick one of the different templates available (see Figure 13-2). You create a project that you can easily modify and adapt to create your own app.

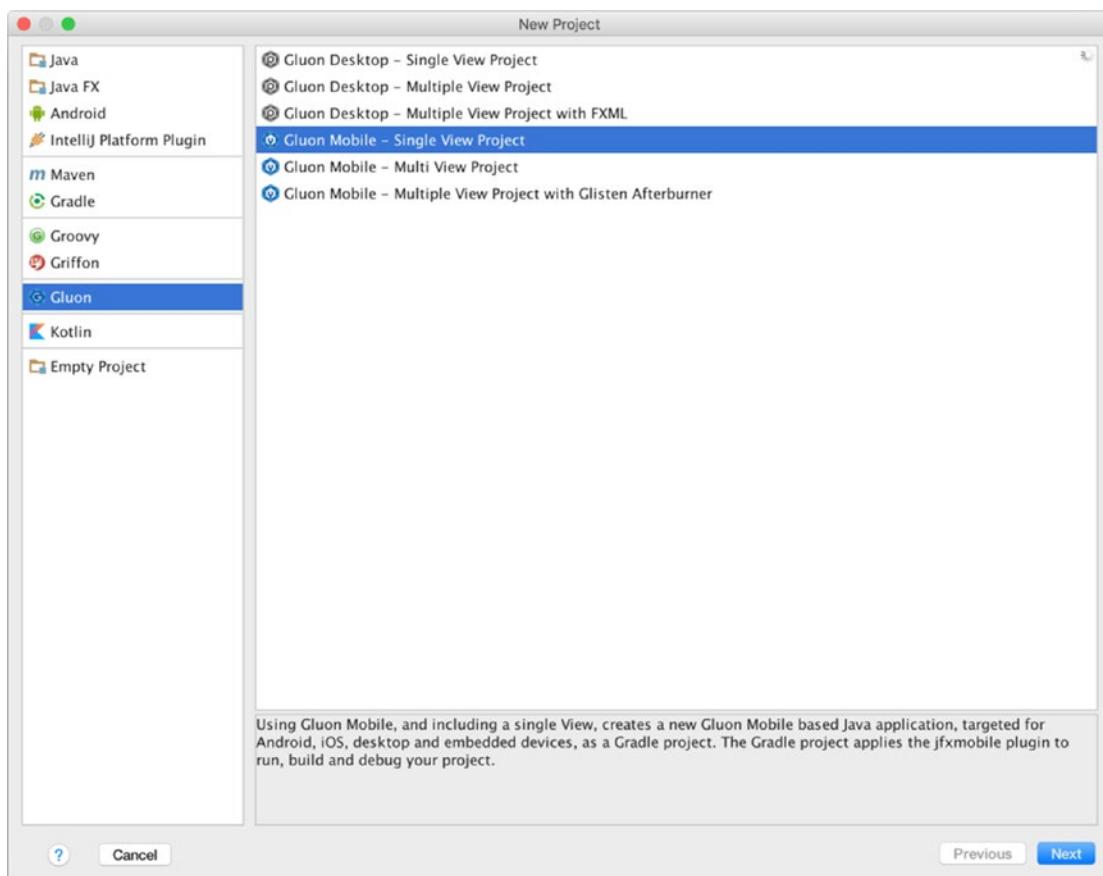


Figure 13-2. Gluon plug-in for IntelliJ and available templates

Charm Glisten

The base class is `MobileApplication`, and it extends from `Application`. It does not require any specific methods to be implemented, but typically `init` can be overridden to register the different views, and `postInit` can access the scene and apply an style sheet, for instance.

Glisten user interfaces are built using views. A *view* is a Glisten container that allows you to add nodes to its top, center, and bottom.

Usually a `View` instance is created by providing a name for the view and the node for its content. This instance is added to a factory of views so the Glisten UI can load and unload them on demand.

By default, the first view displayed when the stage is shown is called `Home View`. It has no predefined content, so this view has to be designed by the developer, but its name is already assigned: `MobileApplication.HOME_VIEW`. The short snippet in Listing 13-6 will create a very simple mobile application with a single view, as shown in Figure 13-3.

By default, `glisten.css` is the style sheet added to a Gluon Mobile application. It is based on Material Design from Google (<https://material.io>).

See the documentation at http://docs.gluonhq.com/charm/latest/#_charm_glisten to find out more about the Glisten controls.

Listing 13-6. Creating a Home View

```
import com.gluonhq.charm.glisten.application.MobileApplication;
import com.gluonhq.charm.glisten.mvc.View;
import com.gluonhq.charm.glisten.visual.Swatch;
import javafx.scene.Scene;
import javafx.scene.control.Button;

public class MyApp extends MobileApplication {

    public static final String BASIC_VIEW = HOME_VIEW;

    @Override
    public void init() {
        addViewFactory(BASIC_VIEW, () -> new View(new Button("Hello Glisten!")));
    }

    @Override
    public void postInit(Scene scene) {
        Swatch.BLUE.assignTo(scene);
    }
}
```

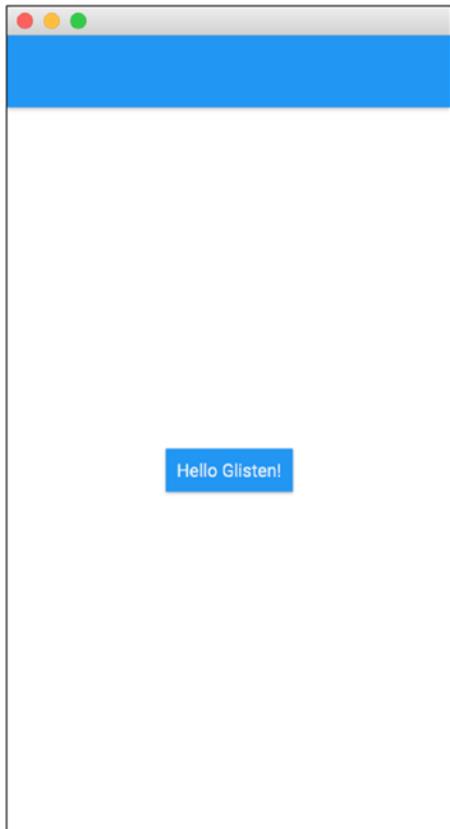


Figure 13-3. MyApp running on the desktop

License

While JavaFXPorts, Charm Down, or Charm Connect are fully open source projects, Gluon Mobile is a commercial project and requires a license. As you can see at <http://gluonhq.com/products/mobile/buy>, there is a free tier and the library can be used for testing with the 100% of its features, without a license.

Example: The BasketStats App

This example shows you how to create, step by step, a full Java mobile app that you can deploy to your mobile devices. You can use it to keep track of the annotation during the basketball games of your kids, for instance.

In this case, you'll use NetBeans and the Gluon plug-in for NetBeans.

Creating the Project

Let's create a new project using the Gluon plug-in. In NetBeans, choose File ▶ New Project and select Gluon on the left. Select Gluon Mobile – Glisten -Afterburner Project from the list of available projects, as shown in Figure 13-4.

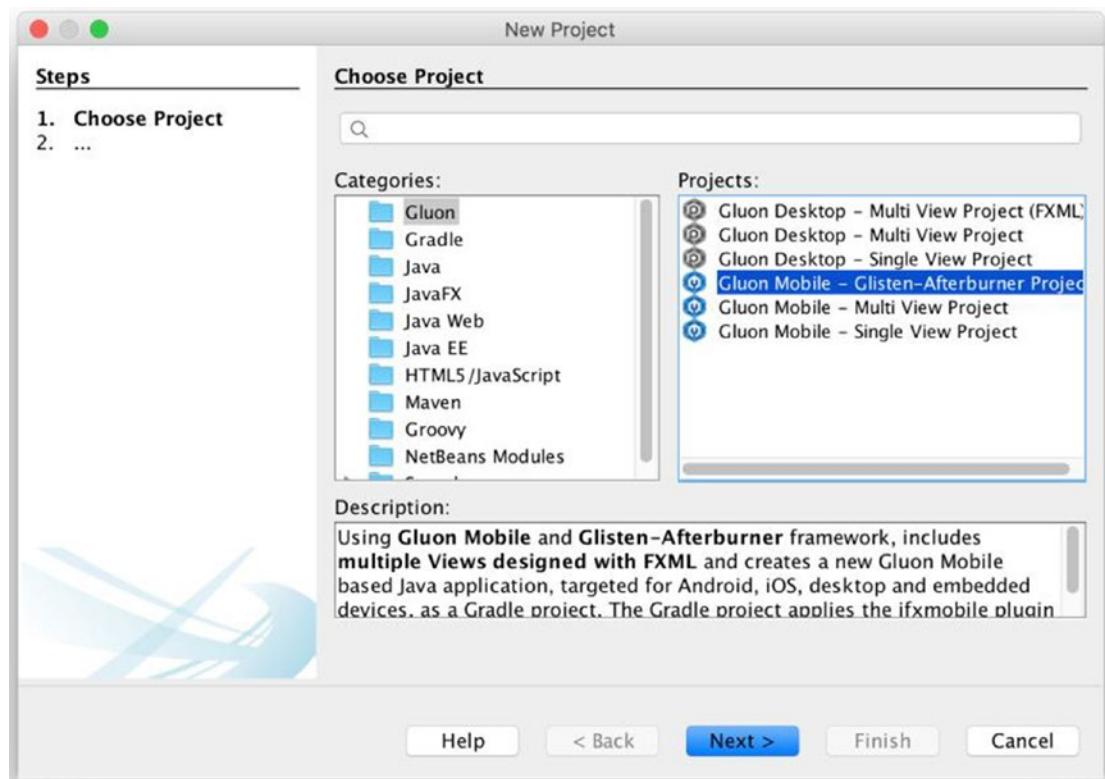


Figure 13-4. New project: Choose project

Add a proper name to the application (BasketStats), find a proper location, add the package name, and change the main class name if required, as shown in Figure 13-5.

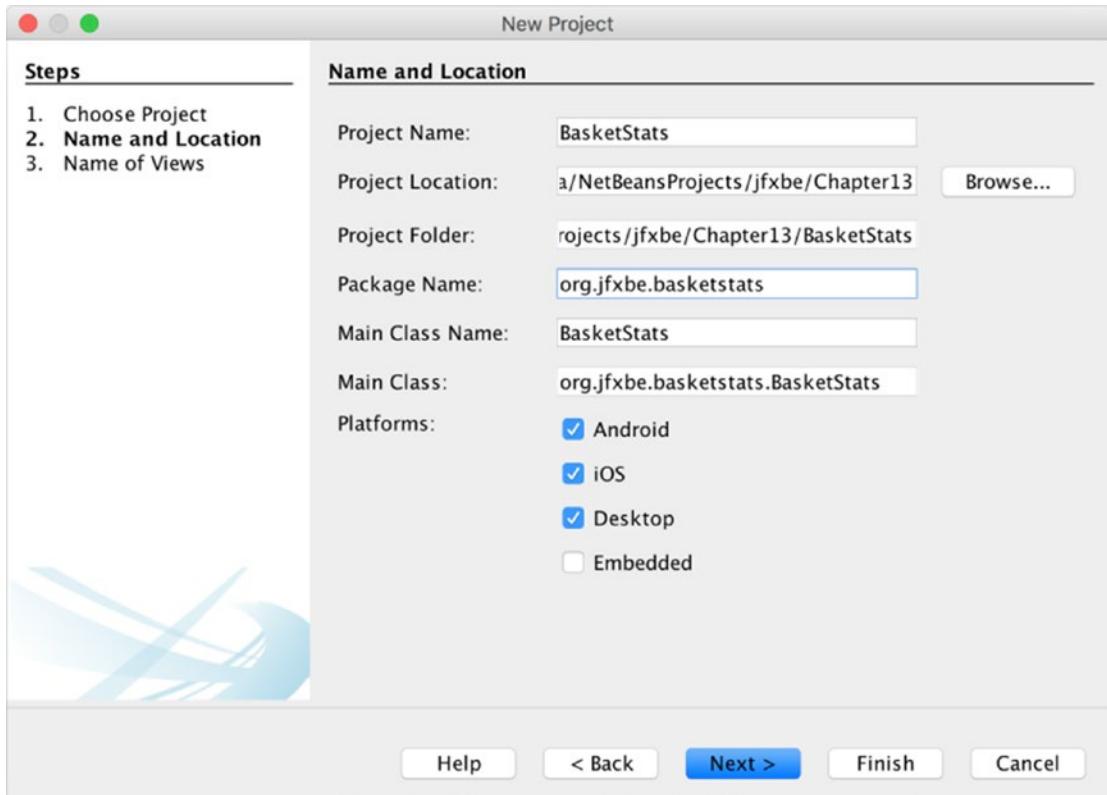


Figure 13-5. New project: Set name and location

Click Next and set these names—Main for the primary view and Board for the secondary view—as shown in Figure 13-6. Click Finish and the project will be created.

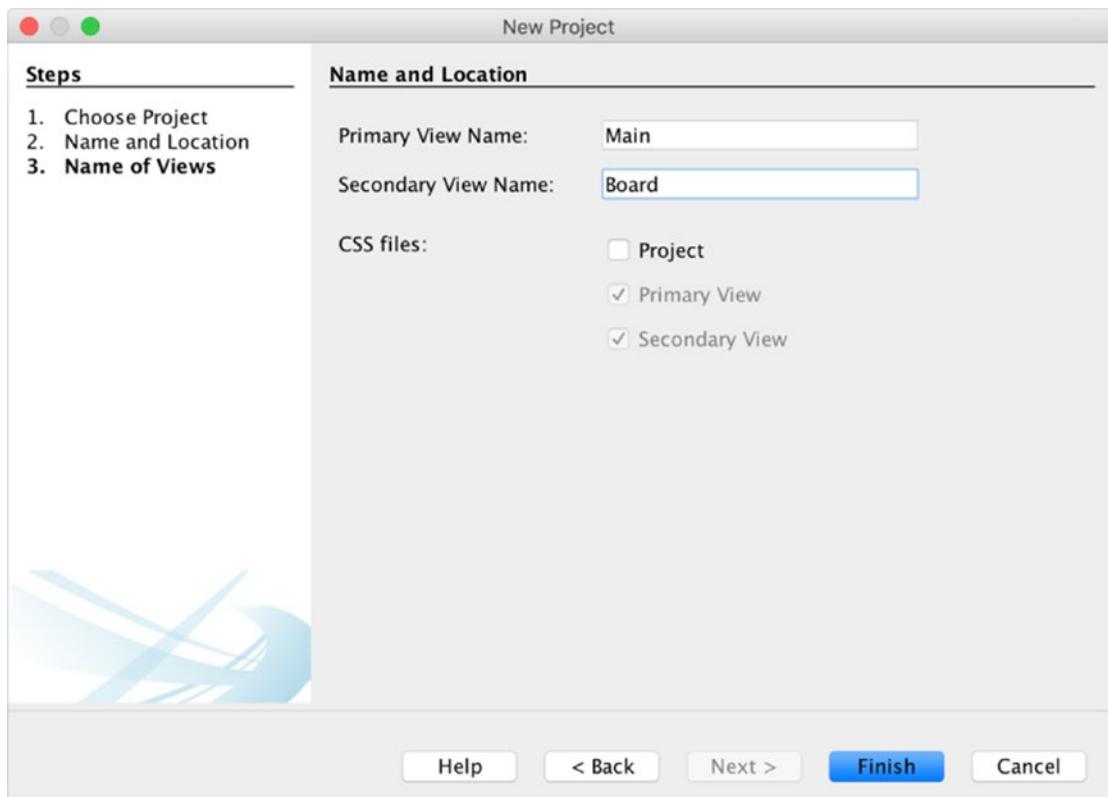


Figure 13-6. New project: Set views names

The full structure of the project can be seen in Figure 13-7. Listing 13-7 contains the build.gradle file, showing the versions available at the time of this writing.

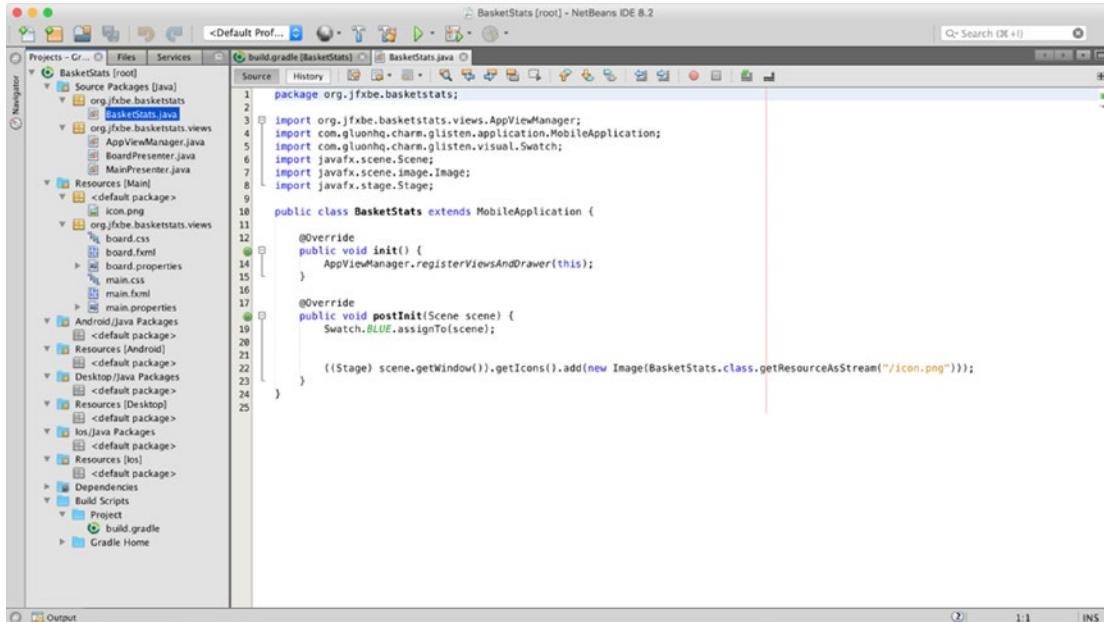


Figure 13-7. Project created: The MobileApplication class

Listing 13-7. Default build.gradle File

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'org.javafxports:jfxmobile-plugin:1.3.5'
    }
}

apply plugin: 'org.javafxports.jfxmobile'

repositories {
    jcenter()
    maven {
        url 'http://nexus.gluonhq.com/nexus/content/repositories/releases'
    }
}
```

```

mainClassName = 'org.jfxbe.basketstats.BasketStats'

dependencies {
    compile 'com.gluonhq:glisten-afterburner:1.2.0'
    compile 'com.gluonhq:charm-glisten:4.3.5'
}

jfxmobile {
    downConfig {
        version = '3.3.0'
        plugins 'display', 'lifecycle', 'statusbar', 'storage'
    }
    android {
        manifest = 'src/android/AndroidManifest.xml'
    }
    ios {
        infoPList = file('src/ios/Default-Info.plist')
        forceLinkClasses = [
            'org.jfxbe.basketstats.*.*',
            'com.gluonhq.*.*',
            'javax.annotations.*.*',
            'javax.inject.*.*',
            'javax.json.*.*',
            'org.glassfish.json.*.*'
        ]
    }
}

```

In the application class, the `init` method is overridden to create and register the views. Listing 13-8 contains the `AppViewManager` class that takes care of adding the views to an `AppViewRegistry` instance. The views are created with FXML with the `AppView` class, using the Afterburner framework (<https://github.com/AdamBien/afterburner.fx>). A `NavigationDrawer` control is also created and registered. Figure 13-8 shows this layer when you run the project.

Listing 13-8. The `AppViewManager` Class

```

package org.jfxbe.basketstats.views;

import com.gluonhq.charm.glisten.afterburner.AppView;
import static com.gluonhq.charm.glisten.afterburner.AppView.Flag.HOME_VIEW;
import static com.gluonhq.charm.glisten.afterburner.AppView.Flag.SHOW_IN_DRAWER;
import static com.gluonhq.charm.glisten.afterburner.AppView.Flag.SKIP_VIEW_STACK;
import com.gluonhq.charm.glisten.afterburner.AppViewRegistry;
import com.gluonhq.charm.glisten.afterburner.GluonPresenter;
import com.gluonhq.charm.glisten.afterburner.DefaultDrawerManager;

```

```
import com.gluonhq.charm.glisten.application.MobileApplication;
import com.gluonhq.charm.glisten.control.Avatar;
import com.gluonhq.charm.glisten.control.NavigationDrawer;
import com.gluonhq.charm.glisten.visual.MaterialDesignIcon;
import javafx.scene.image.Image;
import java.util.Locale;
import org.jfxbe.basketstats.BasketStats;

public class AppViewManager {

    public static final AppViewRegistry REGISTRY = new AppViewRegistry();

    public static final AppView MAIN_VIEW = view("Game Manager", MainPresenter.class,
        MaterialDesignIcon.GAMES, SHOW_IN_DRAWER, HOME_VIEW, SKIP_VIEW_STACK);
    public static final AppView BOARD_VIEW = view("Board", BoardPresenter.class,
        MaterialDesignIcon.DASHBOARD, SHOW_IN_DRAWER);

    private static AppView view(String title,
        Class<? extends GluonPresenter<?>> presenterClass,
        MaterialDesignIcon menuIcon, AppView.Flag... flags ) {
        return REGISTRY.createView(name(presenterClass), title, presenterClass,
            menuIcon, flags);
    }

    private static String name(Class<? extends GluonPresenter<?>> presenterClass) {
        return presenterClass.getSimpleName().toUpperCase(Locale.ROOT)
            .replace("PRESENTER", "");
    }

    public static void registerViewsAndDrawer(MobileApplication app) {
        for (AppView view : REGISTRY.getViews()) {
            view.registerView(app);
        }

        NavigationDrawer.Header header = new NavigationDrawer.Header("Gluon Mobile",
            "The BasketStats App",
            new Avatar(21,
                new Image(BasketStats.class.getResourceAsStream("/icon.png"))));

        DefaultDrawerManager drawerManager = new DefaultDrawerManager(app,
            header, REGISTRY.getViews());
        drawerManager.installDrawer();
    }
}
```

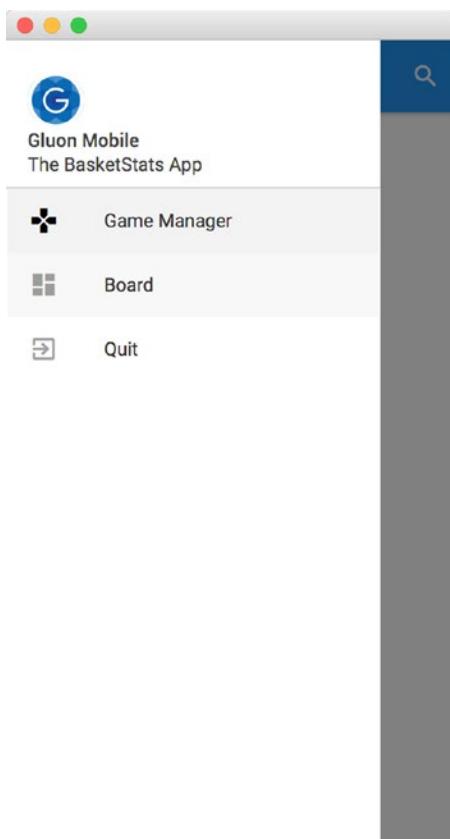


Figure 13-8. Running the project on desktop

Adding the Model

Let's add the model now. First, you define a `GameEvent` class with some primitives like the annotated score, the date time of the event, the number of periods, and the number of the team. See Listing 13-9.

Listing 13-9. The GameEvent Class

```
package org.jfxbe.basketstats.model;

import java.time.LocalDateTime;
import java.time.ZoneOffset;

public class GameEvent {

    private int score;
    private long dateTime;
    private int period;
    private int team;
    private String partialScore;
```

```
/**  
 * Game event when a team scores  
 * @param score 0, 1, 2, 3. 0 means other game event  
 * @param dateTime  
 * @param period 1 to 4, 5 means game ended  
 * @param team 1 for team A, 2 for team B, 0 for other game event  
 */  
public GameEvent(int score, long dateTime, int period, int team) {  
    this.score = score;  
    this.dateTime = dateTime;  
    this.period = period;  
    this.team = team;  
}  
  
/**  
 * Game event when the game starts a period or ends  
 * @param dateTime  
 * @param period  
 */  
public GameEvent(LocalDateTime dateTime, int period) {  
    this(0, dateTime.toInstant(ZoneOffset.UTC).getEpochSecond(), period, 0);  
}  
  
public GameEvent() {  
    this(0, 0, 0, 0);  
}  
  
public int getScore() {  
    return score;  
}  
  
public void setScore(int score) {  
    this.score = score;  
}  
  
public long getDateTime() {  
    return dateTime;  
}  
  
public LocalDateTime getLocalDateTime() {  
    return LocalDateTime.ofEpochSecond(dateTime, 0, ZoneOffset.UTC);  
}  
  
public void setDateDateTime(long dateTime) {  
    this.dateTime = dateTime;  
}  
  
public int getPeriod() {  
    return period;  
}
```

```

public void setPeriod(int period) {
    this.period = period;
}

public int getTeam() {
    return team;
}

public void setTeam(int team) {
    this.team = team;
}

public String getPartialScore() {
    return partialScore;
}

public void setPartialScore(String partialScore) {
    this.partialScore = partialScore;
}

}

```

Now you'll have a Game class with some JavaFX properties like the name and score of the teams, the local date and time of the game, and a list of game events. See Listing 13-10.

Listing 13-10. The Game Class

```

package org.jfxbe.basketstats.model;

import java.time.LocalDateTime;
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.ListProperty;
import javafx.beans.property.ObjectProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.property.SimpleListProperty;
import javafx.beans.property.SimpleObjectProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;

public class Game {

    private final ListProperty<GameEvent> gameEvents;

    public Game() {
        gameEvents = new SimpleListProperty<>(FXCollections.observableArrayList());
    }
}

```

```

private final IntegerProperty scoreA = new SimpleIntegerProperty(this, "scoreA", 0);
public final IntegerProperty scoreAProperty() { return scoreA; }
public final int getScoreA() { return scoreA.get(); }
public final void setScoreA(int value) { scoreA.set(value); }

private final IntegerProperty scoreB = new SimpleIntegerProperty(this, "scoreB", 0);
public final IntegerProperty scoreBProperty() { return scoreB; }
public final int getScoreB() { return scoreB.get(); }
public final void setScoreB(int value) { scoreB.set(value); }

private final StringProperty teamA = new SimpleStringProperty(this, "teamA", "");
public final StringProperty teamAProperty() { return teamA; }
public final String getTeamA() { return teamA.get(); }
public final void setTeamA(String teamA) { this.teamA.set(teamA); }

private final StringProperty teamB = new SimpleStringProperty(this, "teamB", "");
public final StringProperty teamBProperty() { return teamB; }
public final String getTeamB() { return teamB.get(); }
public final void setTeamB(String teamB) { this.teamB.set(teamB); }

private final ObjectProperty<LocalDateTime> localDateTime =
    new SimpleObjectProperty<>(this, "localDate", LocalDateTime.now());
public final ObjectProperty<LocalDateTime> localDateTimeProperty() {
    return localDateTime; }
public final LocalDateTime getLocalDateTime() { return localDateTime.get(); }
public final void setLocalDateTime(LocalDateTime localDateTime) {
    this.localDateTime.set(localDateTime); }

public ListProperty<GameEvent> gameEventsProperty() { return gameEvents; }
public ObservableList<GameEvent> getGameEvents() { return gameEvents.get(); }
public void setGameEvents(ObservableList<GameEvent> gluonGame) {
    gameEvents.set(gluonGame); }

public final String getGameName() {
    return "Game-" + teamA.get() + "-" + teamB.get() + "-" +
        localDateTime.get().toLocalDate().toEpochDay() + "-" +
        localDateTime.get().toLocalTime().toSecondOfDay() + ".gam";
}
}
}

```

Adding the Service

This app will store the games locally, using a file for each game. Now you need to define a Service class to manage this. While you could use any regular file read/file write approach, you'll use Gluon CloudLink Client to persist the games you create in the local storage of the device. While its main purpose is to allow communications with the cloud, you can use it locally as well.

For this, you need a dependency, as shown in Listing 13-11.

Listing 13-11. Required Dependencies

```
dependencies {
    compile 'com.gluonhq:glisten-afterburner:1.2.0'
    compile 'com.gluonhq:charm-glisten:4.3.5'
    compile 'com.gluonhq:charm-cloudlink-client:4.3.5'
}
```

As seen in Listing 13-12, the service will create Game and DataClient instances upon initialization. The latter will use OperationMode.LOCAL_ONLY to indicate only local operations.

The DataClient instance calls the `createListDataReader()` method with the following arguments—an identifier (the unique name of the game), the object class to be read (`GameEvent.class`), and the synchronization flags:

- `SyncFlag.LIST_WRITE_THROUGH`, so changes in the list of game events are automatically stored locally.
- `SyncFlag.OBJECT_WRITE_THROUGH`, so changes in the properties of any game event in the list are also stored locally.

It returns a `ListDataReader` object. Use the static method `DataProvider.retrieveList` with that data reader to obtain a `GluonObservableList<GameEvents>`, which is an observable list of game events that can be used in the different views to get or add new `gameEvents`. These events are immediately added to the local file.

Listing 13-12. The Game Service.

```
package org.jfxbe.basketstats.service;

import com.gluonhq.cloudlink.client.data.DataClient;
import com.gluonhq.cloudlink.client.data.DataClientBuilder;
import com.gluonhq.cloudlink.client.data.OperationMode;
import com.gluonhq.cloudlink.client.data.SyncFlag;
import com.gluonhq.connect.GluonObservableList;
import com.gluonhq.connect.provider.DataProvider;
import javafx.beans.property.ListProperty;
import javax.annotation.PostConstruct;
import org.jfxbe.basketstats.model.Game;
import org.jfxbe.basketstats.model.GameEvent;

public class Service {

    private DataClient dataClient;

    private Game game;

    @PostConstruct
    public void postConstruct() {
        dataClient = DataClientBuilder.create()
```

```

        .operationMode(OperationMode.LOCAL_ONLY)
        .build();

    game = new Game();
}

public GluonObservableList<GameEvent> retrieveGame(String nameGame) {
    game.setGameEvents(null);

    return DataProvider.retrieveList(dataClient.createListDataReader(nameGame,
        GameEvent.class,
        SyncFlag.LIST_WRITE_THROUGH, SyncFlag.OBJECT_WRITE_THROUGH));
}

public void addGameEvent(GameEvent gameEvent) {
    updateScore(gameEvent);
    gameEvent.setPartialScore(":" + game.getScoreA() + " :: " + game.getScoreB());
    game.getGameEvents().add(gameEvent);
}

public Game getGame() {
    return game;
}

public final ListProperty<GameEvent> gameEventsProperty() {
    return game.gameEventsProperty();
}

public void updateScore(GameEvent event) {
    switch (event.getTeam()) {
        case 0:
            if (event.getPeriod() == 1) {
                game.setScoreA(0);
                game.setScoreB(0);
            }
            break;
        case 1:
            game.setScoreA(game.getScoreA() + event.getScore());
            break;
        case 2:
            game.setScoreB(game.getScoreB() + event.getScore());
            break;
    }
}
}

```

You can check the “Data Storage” section of the Gluon CloudLink documentation at http://docs.gluonhq.com/cloudlink/#_data_storage to learn about the `DataManager` and `GluonObservableList` concepts in more detail.

Modifying the Main View

It's now time to modify the default views. Their FXML files can be edited with Scene Builder. Since version 8.3.0, which can be downloaded and installed from <http://gluonhq.com/products/scene-builder/>, the required dependencies are included, and you can easily design Gluon Mobile views. See Figure 13-9.

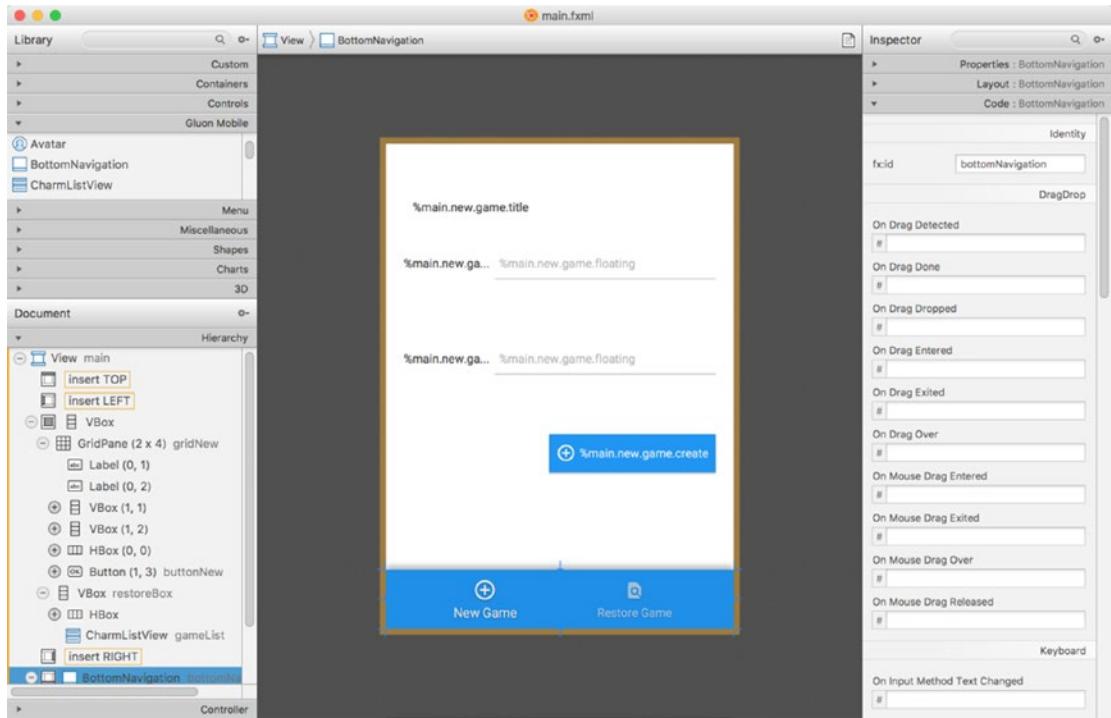


Figure 13-9. Main view edited with Scene Builder

First, open the main view (`main.fxml`). To preview the Gluon style, select Preview ► JavaFX Theme ► Gluon Mobile. Remove the default content and add a `BottomNavigationView` control to the bottom of the view with two buttons: one to create a new game and one to retrieve an existing game.

Now you add a `VBox` container that will hold a `GridPane` with two `TextField` controls from Gluon Mobile, so the user can set the name of the teams. It will be visible when the New Game button is selected. And a second `VBox` that will hold a `CharmListView` control, an enhanced list with headers, to show the different games available. It will be visible when the Restore button is selected. Note that for every control added, you assign an `fx:id` tag, so those can be referenced from the controller class. Figure 13-9 shows the main view once the changes have been added.

The edited FXML file, as the rest of the source code of this example, can be found in the repository of this book.

Now add all the tagged controls to the presenter, as shown in Listing 13-13.

Listing 13-13. The Main Presenter

```
package org.jfxbe.basketstats.views;

import static com.gluonhq.charm.glisten.afterburner.DefaultDrawerManager.DRAWER_LAYER;
import com.gluonhq.charm.glisten.afterburner.GluonPresenter;
import com.gluonhq.charm.glisten.control.*;
import com.gluonhq.charm.glisten.mvc.View;
import com.gluonhq.charm.glisten.visual.MaterialDesignIcon;
import java.time.*;
import java.util.ResourceBundle;
import javafx.collections.FXCollections;
import javafx.fxml.FXML;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javax.inject.Inject;
import org.jfxbe.basketstats.BasketStats;
import org.jfxbe.basketstats.service.Service;
import org.jfxbe.basketstats.utils.GameUtils;
import org.jfxbe.basketstats.views.cells.*;

public class MainPresenter extends GluonPresenter<BasketStats> {

    @Inject private Service service;

    @FXML private View main;
    @FXML private GridPane gridNew;
    @FXML private TextField textNameA, textNameB;
    @FXML private Button buttonNew;
    @FXML private VBox restoreBox;
    @FXML private CharmListView<String, LocalDate> gameList;
    @FXML private ToggleButton newToggle, restoreToggle;
    @FXML private ResourceBundle resources;

    public void initialize() {
        main.showingProperty().addListener((obs, oldValue, newValue) -> {
            if (newValue) {
                AppBar appBar = getApp().getAppBar();
                appBar.setNavIcon(MaterialDesignIcon.MENU.button(e ->
                    getApp().showLayer(DRAWER_LAYER)));
                appBar.setTitleText(resources.getString("main.app.title"));
            }
        });
        buttonNew.disableProperty().bind(textNameA.textProperty().isEmpty()
            .or(textNameB.textProperty().isEmpty())
            .or(textNameA.textProperty().isEqualToString(textNameB.textProperty())));
    }
}
```

```

buttonNew.setOnAction(e -> {
    service.getGame().setTeamA(textNameA.getText());
    service.getGame().setTeamB(textNameB.getText());
    service.getGame().setLocalDateTime(LocalDateTime.now());
    GameUtils.restoreGame(service.getGame().getGameName());
});

restoreBox.managedProperty().bind(restoreBox.visibleProperty());
gridNew.managedProperty().bind(gridNew.visibleProperty());

gameList.setPlaceholder(new Label(resources.getString("main.listview.noitems")));
gameList.setHeadersFunction(GameUtils::getLocalDateFromGame);
gameList.setHeaderCellFactory(p -> new HeaderGameListCell());
gameList.setHeaderComparator((l1, l2) -> l2.compareTo(l1));

gameList.setCellFactory(p -> new GameListCell(resources));

restoreToggle.selectedProperty().addListener((obs, ov, nv) -> {
    if (nv) {
        gridNew.setVisible(false);
        restoreBox.setVisible(true);
        gameList.setItems(FXCollections.observableArrayList(
            GameUtils.retrieveGames()));
    }
});

newToggle.selectedProperty().addListener((obs, ov, nv) -> {
    if (nv) {
        restoreBox.setVisible(false);
        gridNew.setVisible(true);
    }
});
newToggle.setSelected(true);
}

}

```

The presenter adds the required listeners to the controls and cell factories for the header and regular cells of the `CharmListView` control upon initialization (both cell factory classes can be found in the repository of this book). Using injection, the service instance is added to the presenter. To learn more about the Gluon Mobile controls, check out the documentation at <http://docs.gluonhq.com/charm/javadoc/latest/>.

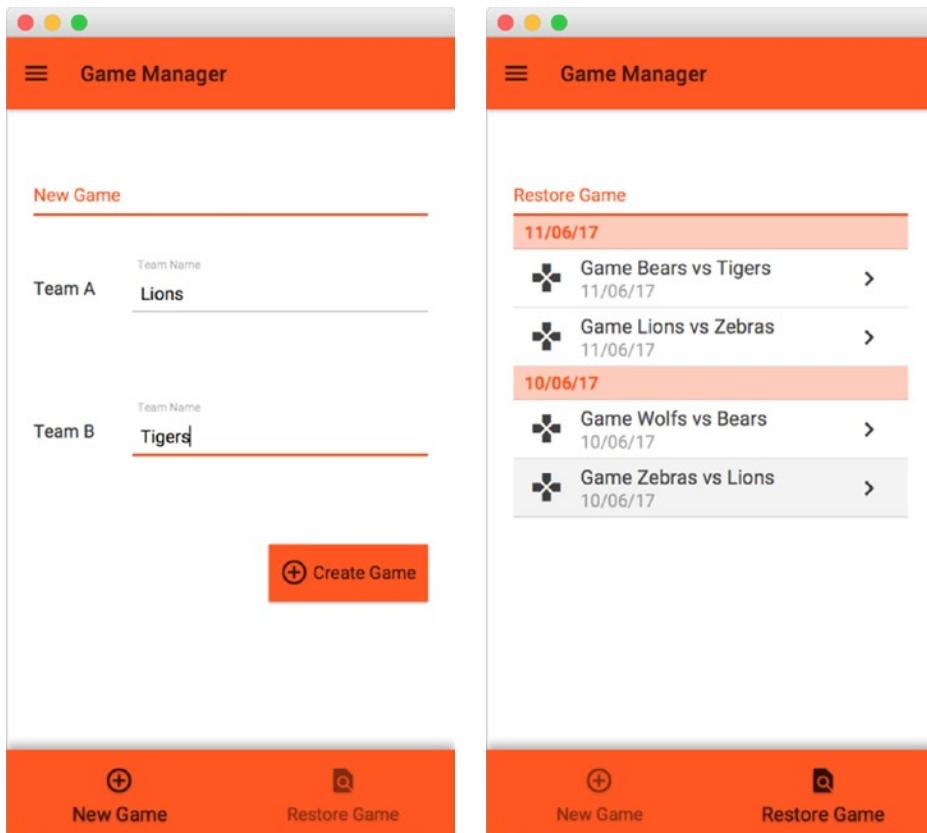
When the New Game toggle is clicked, the grid is visible and the list view is hidden, as shown in Figure 13-10a. When the Restore Game toggle is selected, the grid will be hidden, and the list view will present all the existing games so far, as shown in Figure 13-10b. Note that the stored games have a name and extension like Game-Bears-Tigers-17328-70904.gam, as defined in the `Game` class.

Note the use of the `Storage` service from Charm Down to retrieve the files available in the local folder. In the case of iOS or Android, the returned directory is private and accessible only from the app, as shown in Listing 13-14.

Listing 13-14. Using the Storage Service

```
package org.jfxbe.basketstats.utils;

public class GameUtils {
    public static List<String> retrieveGames() {
        File root = Services.get(StorageService.class)
            .flatMap(storage -> storage.getPrivateStorage())
            .orElseThrow(() -> new RuntimeException("No storage found"));
        List<String> list = new ArrayList<>();
        for (File file : root.listFiles((dir, name) -> name.startsWith("Game")
            && name.endsWith(".gam"))) {
            list.add(file.getName());
        }
        return list;
    }
}
```

**Figure 13-10.** Main view: a) New Game view, b) Restore Game view

Modifying the Board View

Now you'll see how to modify the board view. Open `board.fxml` with Scene Builder, remove the default content, and add to the center of the view a Gluon Mobile `ExpansionPanelContainer` control with three `ExpansionPanel` controls:

- In the first one, its `ExpandedPanel` will hold a `GridPane` with the current annotation and period and the buttons to annotate one, two, or three points for each team. `CollapsedPanel` will hold a label with the current period and score.
- The second one will have in the expanded panel a `CharmListView` control with the game events: a period started, one team score, and some points. The `CollapsedPanel` will hold a label.
- The third one will have in the expanded panel a `LinearChart` control with the game evolution. The `CollapsedPanel` will hold a label.

Figure 13-11 shows the modified view in Scene Builder. Note that every control required in the presenter has to define an `fx:id` tag. The edited FXML file can be found in the repository of this book.

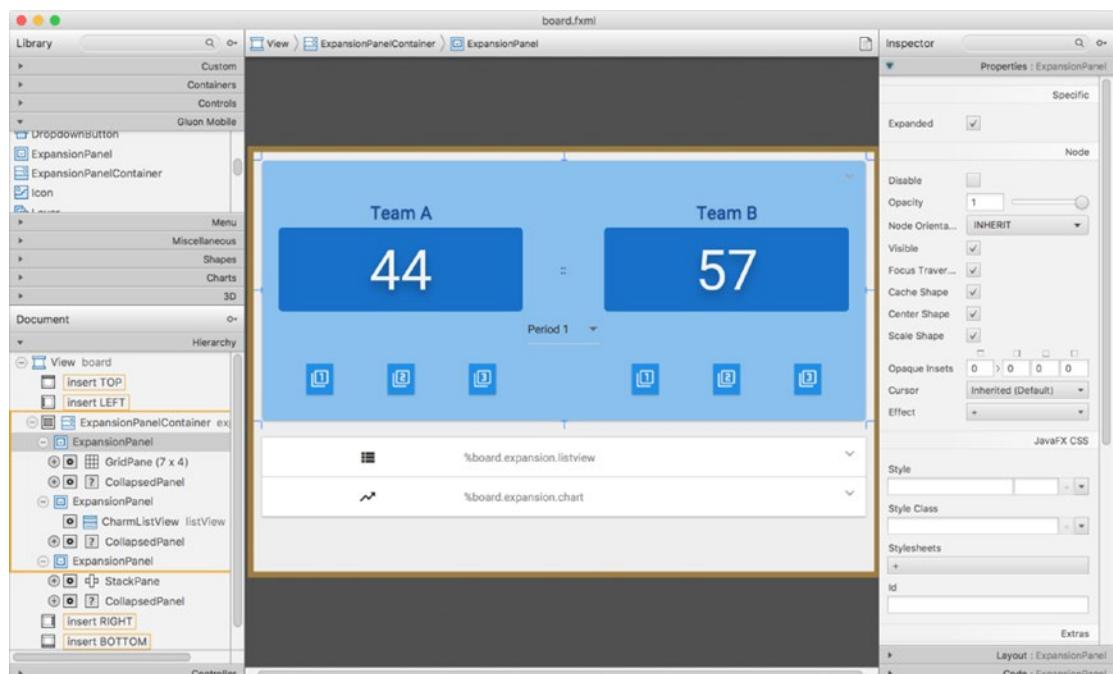


Figure 13-11. Board view edited with Scene Builder

Now you'll add all the tagged controls to the presenter, as shown in Listing 13-15. The factory cells classes can be found in the repository of this book. You'll find a link to that repository from the book's catalog page on the apress.com web site.

Listing 13-15. The Board Presenter

```
package org.jfxbe.basketstats.views;

import static com.gluonhq.charm.glisten.afterburner.DefaultDrawerManager.DRAWER_LAYER;
import com.gluonhq.charm.glisten.afterburner.GluonPresenter;
import com.gluonhq.charm.glisten.control.*;
import com.gluonhq.charm.glisten.mvc.View;
import com.gluonhq.charm.glisten.visual.MaterialDesignIcon;
import com.gluonhq.connect.GluonObservableList;
import java.text.MessageFormat;
import java.time.*;
import java.time.format.*;
import java.util.Arrays;
import java.util.List;
import java.util.ResourceBundle;
import javafx.beans.binding.Bindings;
import javafx.beans.property.*;
import javafx.fxml.FXML;
import javafx.scene.chart.*;
import javafx.scene.chart.XYChart.Series;
import javafx.scene.control.*;
import javafx.util.StringConverter;
import javax.inject.Inject;
import org.jfxbe.basketstats.BasketStats;
import org.jfxbe.basketstats.model.GameEvent;
import org.jfxbe.basketstats.service.Service;
import org.jfxbe.basketstats.views.cells.*;

public class BoardPresenter extends GluonPresenter<BasketStats> {

    @Inject private Service service;

    @FXML private View board;
    @FXML private ExpansionPanelContainer expansion;
    @FXML private Label labelTeamA, labelTeamB;
    @FXML private Label scoreTeamA, scoreTeamB, globalScore;
    @FXML private Button plusOneTeamA, plusTwoTeamA, plusThreeTeamA;
    @FXML private Button plusOneTeamB, plusTwoTeamB, plusThreeTeamB;
    @FXML private DropdownButton dropdown;
    @FXML private MenuItem period1, period2, period3, period4;
    @FXML private CharmListView<GameEvent, Integer> listView;
    @FXML private LineChart<Number, Number> chart;
    @FXML private NumberAxis xAxis, yAxis;
    @FXML private ResourceBundle resources;
```

```

private Button buttonStart, buttonStop;
private List<Button> buttons;
private List<MenuItem> periods;
private final IntegerProperty period = new SimpleIntegerProperty();
private Series<Number, Number> teamASeries, teamBSeries;

public void initialize() {
    buttons = Arrays.asList(plusOneTeamA, plusTwoTeamA, plusThreeTeamA,
                           plusOneTeamB, plusTwoTeamB, plusThreeTeamB);
    periods = Arrays.asList(period1, period2, period3, period4);

    enableGame(false);

    buttonStart = MaterialDesignIcon.PLAY_CIRCLE_OUTLINE
                  .button(e -> startGame());
    buttonStop = MaterialDesignIcon.STOP.button(e -> stopGame());
    buttonStart.setDisable(true);
    buttonStop.setDisable(true);

    board.showingProperty().addListener((obs, oldValue, newValue) -> {
        if (newValue) {
            AppBar appBar = getApp().getAppBar();
            appBar.setNavIcon(MaterialDesignIcon.MENU.button(e ->
                getApp().showLayer(DRAWER_LAYER)));
            appBar.setTitleText(resources.getString("board.app.title"));
            appBar.getActionItems().addAll(buttonStart, buttonStop);
        }
    });

    for (ExpansionPanel panel : expansion.getItems()) {
        panel.expandedProperty().addListener((obs, ov, nv) -> {
            if (nv) {
                for (ExpansionPanel otherPanel : expansion.getItems()) {
                    if (!otherPanel.equals(panel)) {
                        otherPanel.setExpanded(false);
                    }
                }
            }
        });
    }

    period2.setOnAction(e -> addPeriodEvent(2));
    period3.setOnAction(e -> addPeriodEvent(3));
    period4.setOnAction(e -> addPeriodEvent(4));

    plusOneTeamA.setOnAction(e -> addScoreEvent(1, 1));
    plusTwoTeamA.setOnAction(e -> addScoreEvent(2, 1));
    plusThreeTeamA.setOnAction(e -> addScoreEvent(3, 1));
    plusOneTeamB.setOnAction(e -> addScoreEvent(1, 2));
    plusTwoTeamB.setOnAction(e -> addScoreEvent(2, 2));
    plusThreeTeamB.setOnAction(e -> addScoreEvent(3, 2));
}

```

```

labelTeamA.textProperty().bind(service.getGame().teamAProperty());
labelTeamB.textProperty().bind(service.getGame().teamBProperty());
scoreTeamA.textProperty().bind(service.getGame().scoreAProperty().asString());
scoreTeamB.textProperty().bind(service.getGame().scoreBProperty().asString());
globalScore.textProperty().bind(Bindings.createStringBinding(() -> {
    String p;
    if (period.get() < 5) {
        p = MessageFormat.format(resources.getString("board.expansion.score"),
            period.get());
    } else {
        p = resources.getString("board.listview.end");
    }
    return String.format("(%s) %s :: %s", p, scoreTeamA.getText(),
        scoreTeamB.getText());
}, period, scoreTeamA.textProperty(), scoreTeamB.textProperty()));

// CharmListView
listView.setHeadersFunction(GameEvent::getPeriod);
listView.setPlaceholder(new Label(resources.getString("board.listview.noitems")));

listView.setComparator((GameEvent e1, GameEvent e2) ->
    Long.compare(e1.getDateTime(), e2.getDateTime()));
listView.setHeaderCellFactory(p -> new HeaderGameEventListCell(resources));
listView.setCellFactory(p -> new GameEventListCell(resources));
listView.setItems(service.gameEventsProperty());

// Chart
xAxis.setAutoRanging(true);
xAxis.setForceZeroInRange(false);
xAxis.setTickLabelFormatter(new StringConverter<Number>() {
    @Override
    public String toString(Number t) {
        return DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM)
            .format(LocalDateTime.ofEpochSecond(t.longValue(), 0,
                ZoneOffset.UTC));
    }
    @Override
    public Number fromString(String string) {
        throw new UnsupportedOperationException("Not supported yet.");
    }
});
teamASeries = new Series<>();
teamASeries.setName(service.getGame().getTeamA());
teamBSeries = new Series<>();
teamBSeries.setName(service.getGame().getTeamB());
chart.getData().addAll(teamASeries, teamBSeries);
}

private void startGame() {
    buttonStart.setDisable(true);
    buttonStop.setDisable(false);
}

```

```
enableGame(true);
addPeriodEvent(1);
}

private void stopGame() {
    buttonStop.setDisable(true);
    enableGame(false);
    addPeriodEvent(5);
}

private void addPeriodEvent(int period) {
    addPeriodEvent(period, LocalDateTime.now());
}

private void addPeriodEvent(int period, LocalDateTime time) {
    this.period.set(period);
    service.addGameEvent(new GameEvent(time, period));
    for (int i = 0; i < 4; i++) {
        periods.get(i).setDisable(i != period);
    }
}

private void addScoreEvent(int score, int team) {
    addScoreEvent(score, team, LocalDateTime.now());
}

private void addScoreEvent(int score, int team, LocalDateTime time) {
    final GameEvent gameEvent = new GameEvent(score,
        time.toInstant(ZoneOffset.UTC).getEpochSecond(), period.get(), team);
    service.addGameEvent(gameEvent);
    if (team == 1) {
        teamASeries.getData().add(
            new XYChart.Data<>(gameEvent.getDate(), 
                service.getGame().getScoreA()));
    } else {
        teamBSeries.getData().add(
            new XYChart.Data<>(gameEvent.getDate(), 
                service.getGame().getScoreB()));
    }
}

private void enableGame(boolean value) {
    dropdown.setDisable(!value);
    for (Button button : buttons) {
        button.setDisable(!value);
    }
}

public void restoreGame(String gameId) {
    teamASeries.getData().clear();
    teamBSeries.getData().clear();
    final String[] split = gameId.split("-");
}
```


The presenter adds the required listeners to the controls and cell factories for the header and regular cells of the `CharmListView` control upon initialization (the cell factory classes are available in the book's repository).

When the Start button is clicked, the grid is enabled, period 1 is set, and the game starts. Now the user can click on the 1, 2, 3 buttons from team A or B whenever there is a basket from any of the teams, or select the next period. Finally, when the user clicks the Stop button, the game ends, as shown in Figure 13-12a.

At any moment, the user can expand any of the other two expansion panels to see a list of events (see Figure 13-12b) or a chart with the evolution of the game (see Figure 13-12c).

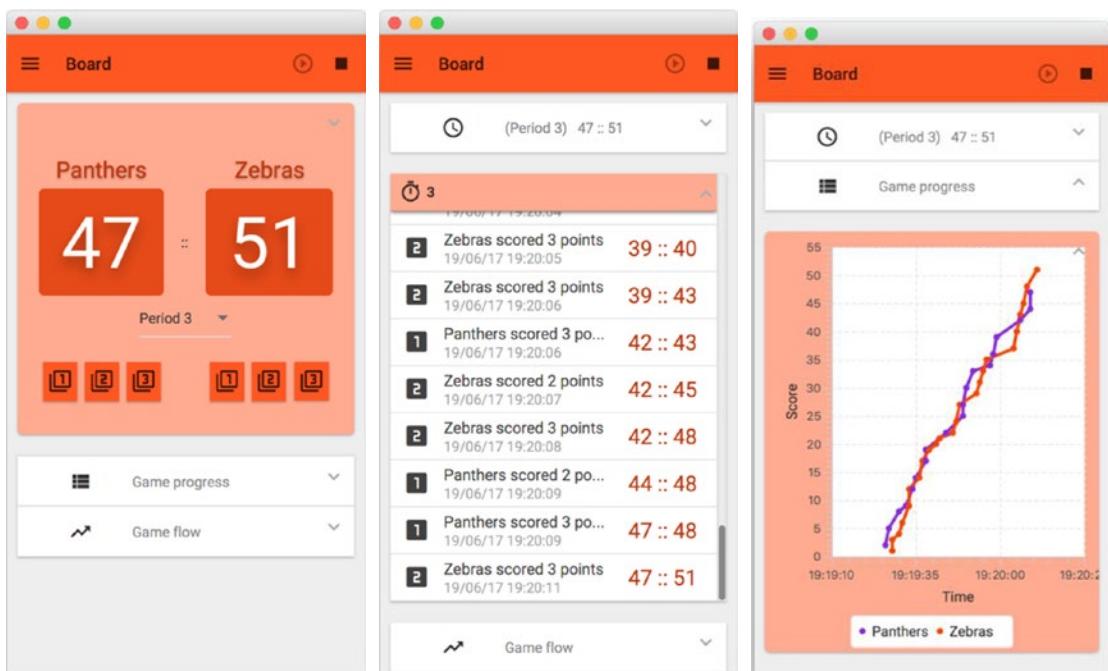


Figure 13-12. Board view: a) Game annotation view, b) Game events view, c) Game evolution view

Deploy to Mobile

Once you have accomplished all the previous steps, it is time to build the app and deploy it on your mobile devices, to test it and check its performance.

Right-click on the root project, and from the context menu, select Tasks ► Android ► androidInstall to deploy on Android or or Tasks ► Launch ► launchIOSDevice to deploy on iOS devices. Figure 13-13 shows the app deployed on an iPad.



Figure 13-13. App deployed on an iPad

More Examples

For more examples of using JavaFX on mobile with Gluon Mobile, go to:

- <http://gluonhq.com/support/samples/>
- <https://github.com/gluonhq/gluon-samples>

For any questions related to the development of mobile apps with JavaFXPorts or Gluon Mobile, go to the StackOverflow forums:

- <https://stackoverflow.com/tags/gluon/>
- <https://stackoverflow.com/tags/gluon-mobile/>
- <https://stackoverflow.com/tags/javafxports/>

Summary

In the first part of this chapter, you were introduced to JavaFXPorts, an open source project that allows Java/JavaFX developers to port their applications to Android and iOS.

In the second part of this chapter, you discovered the Gluon Mobile library. Starting from a default project created by the Gluon plug-in for your IDE, the BasketStats app was developed. With the same codebase and running on desktop, Android, and iOS devices, it showed how to deal with FXML views using Scene Builder, or how to create and inject a service that deals with local storage. Different JavaFX and Gluon Mobile controls were used throughout the example, showing how easy it is to use the library, and how fast the development of mobile apps can be. The use of other open source projects, such as Charm Down, make it easy to use native services on the device (storage device in this example) with a platform-agnostic API.

CHAPTER 14



JavaFX and Gestures

User interfaces are becoming increasingly less mouse-centric, in favor of multi-touch and even touchless input. Gestures are one of the ways humans can communicate with machines naturally, as the latter begin to understand human body language, thanks to complex mathematical algorithms for gesture recognition.

In this chapter, you learn about two new approaches to gesture recognition: In the first part of the chapter, you learn about developing multi-touch JavaFX applications using a touch-enabled device such as the Windows 8 Surface. In the second part of this chapter, you discover an awesome gadget, the Leap Motion device, that provides a touchless approach to developing enhanced JavaFX applications.

In this chapter, the following examples demonstrate JavaFX's touch events API and the Leap Motion Gestures API.

- Animating a shape along a path using touch events (Windows 8)
- Touch, rotate, and zoom in JavaFX 3D (Windows 8)
- A 3D hand model with the skeletal tracking model from Leap Motion

Before getting into the examples, let's look at JavaFX's touch events.

Recognizing Gestures in Your Application

Touch-enabled devices are currently part of our culture. From smartphones to tablets, we find ourselves immersed in software. In this section, you learn the JavaFX `TouchEvent` API for touch-enabled devices. The examples in this section are executed on the Windows 8 surface.

If you are familiar with gestures and gesture programming in general, then you can skip to Table 14-1, which lists all the gesture events currently supported by JavaFX 9. If you are new to gesture programming, first let's discuss the differences between a `MouseEvent` and `GestureEvent`.

Table 14-1. Gesture Events

Gesture Event	Description
ROTATION_STARTED	Two touch points recognized, with one touch point initiating a circular movement.
ROTATE	Two touch points recognized, with one or both touch points performing a circular movement.
ROTATE_FINISHED	Two touch points recognized, with both touch points halted after performing a circular movement.
SCROLL_STARTED	One or more touch points that have begun to move in a slow horizontal or vertical sliding movement.
SCROLL	One or more touch points that are moving in a slow horizontal or vertical sliding movement.
SCROLL_FINISHED	One or more touch points that have ceased moving in a slow horizontal or vertical sliding movement.
SWIPE_LEFT	One or more touch points that are moving in a quick sliding movement to the left edge of the screen.
SWIPE_DOWN	One or more touch points that are moving in a quick sliding movement to the bottom edge of the screen.
SWIPE_RIGHT	One or more touch points that are moving in a quick sliding movement to the right edge of the screen.
SWIPE_UP	One or more touch points that are moving in a quick sliding movement to the top edge of the screen.
TOUCH_PRESSED	A single touch point recognized.
TOUCH_STATIONARY	A single touch point that has remained in the same relative screen location without movement.
TOUCH_MOVED	A single touch point that has moved from a previously recognized TOUCH_PRESSED or TOUCH_STATIONARY location.
TOUCH_RELEASED	A single touch point that has been lifted from the screen.
ZOOM_STARTED	Two touch points recognized, initiating a pinch or stretch motion.
ZOOM	Two touch points recognized, performing a pinch or stretch motion.
ZOOM_FINISHED	Two touch points recognized, with both touch points halted after performing a pinch or stretch motion.

Traditional MouseEvent programming is based on a simple 2D coordinate system aligned with the pixels of the screen. Typically mouse event handling is oriented toward wherever the mouse is at the moment of the event. Gestures are slightly different in that they are oriented toward touch points and their relative motion. A *touch point* is the location on the screen or device that detects interaction through physical contact. There can be multiple touch points simultaneously, and each touch point instance contains information such as X and Y coordinates, GUI component picking, and grouping of related touch points. Generally touch points are treated in a similar fashion as mouse clicks.

The code for the first example of this chapter provided by Listing 14-1 shows the mouse and touch point event handling side by side. The example explains what the gesture TouchPoint equivalents are to various mouse events such as MousePressed, MouseDragged, and MouseReleased.

Touch points are the basis of the more useful gestures such as rotations, scrolls, swipes, and zooms. These more complex gestures involve multiple touch points and movement or inertia associated with the touch points. For instance, touching and dragging your finger across a touch screen will simply register multiple touch point events in a similar fashion to `MouseEvent.MOUSE_DRAGGED`. However, a registered `TouchPoint` followed by a quick swipe to the edge of the device translates to a swipe gesture. This expands the toolset of the developer by providing a multitude of additional interface options that generally don't exist within a standard mouse and keyboard GUI.

Gesture events are enabled for your application in the same manner as all event processing: via event handlers. Table 14-1 provides a listing of the available gesture events.

Example: Animating Shapes Along a Path Using Touch Events

Let's say you want to allow the user of your application to trace a path within your visual application. Tracing through a visual presentation is a very natural interaction for humans. A simple example might be to record the movement of a mouse on the screen and then "play back" the movement through a smooth JavaFX animation. Listing 14-1 shows a complete example of this. Listing 14-2 contains the Java 9 module definition.

Listing 14-1. Animating a Circle Using Touch Events

```
package com.jfxbe.touchevents;

import javafx.animation.PathTransition;
import javafx.application.Application;
import javafx.geometry.Point2D;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.paint.CycleMethod;
import javafx.scene.paint.RadialGradient;
import javafx.scene.paint.Stop;
import javafx.scene.shape.Circle;
import javafx.scene.shape.LineTo;
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.Path;
import javafx.stage.Stage;
import javafx.util.Duration;

/**
 * @author cdea
 */
public class TouchEvents extends Application {

    private final Path onePath = new Path();
    private Point2D anchorPt;

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Chapter 17 Touch Events");
    }
}
```

```

Group root = new Group(onePath);

Scene scene = new Scene(root, 600, 800, Color.WHITE);

RadialGradient gradient1 = new RadialGradient(0, 0.1,
    100, 100,
    20,
    false,
    CycleMethod.NO_CYCLE,
    new Stop(0, Color.RED),
    new Stop(1, Color.BLACK));

// create a sphere
Circle sphere = new Circle(100, 100, 20, gradient1);

// add sphere
root.getChildren().add(sphere);

// animate sphere by following the path.
PathTransition pathTransition =
    new PathTransition(Duration.millis(4000), onePath, sphere);
pathTransition.setCycleCount(1);
pathTransition.setOrientation(
    PathTransition.OrientationType.ORTHOGONAL_TO_TANGENT);

// once finished clear path
pathTransition.setOnFinished(actionEvent ->
    onePath.getElements().clear());

// starting initial path
scene.setOnMousePressed(mouseEvent ->
    startPath(mouseEvent.getX(), mouseEvent.getY()))
);

scene.setOnTouchPressed(touchEvent ->
    startPath(touchEvent.getTouchPoint().getX(),
        touchEvent.getTouchPoint().getY()));

// dragging creates lineTos added to the path
scene.setOnMouseDragged(mouseEvent ->
    drawPath(mouseEvent.getX(), mouseEvent.getY()));
scene.setOnTouchMoved(touchEvent ->
    drawPath(touchEvent.getTouchPoint().getX(),
        touchEvent.getTouchPoint().getY()));

// end the path when mouse released event
scene.setOnMouseReleased(mouseEvent -> endPath(pathTransition));
scene.setOnTouchReleased(touchEvent -> endPath(pathTransition));

primaryStage.setScene(scene);
primaryStage.show();
}

```

```

private void startPath(double x, double y) {
    onePath.getElements().clear();
    // start point in path
    anchorPt = new Point2D(x, y);
    onePath.setStrokeWidth(3);
    onePath.setStroke(Color.BLACK);
    onePath.getElements()
        .add(new MoveTo(anchorPt.getX(), anchorPt.getY()));
}

private void drawPath(double x, double y) {
    onePath.getElements().add(new LineTo(x, y));
}

private void endPath(PathTransition pathTransition) {
    onePath.setStrokeWidth(0.2);
    if (onePath.getElements().size() > 1) {
        pathTransition.stop();
        pathTransition.playFromStart();
    }
}
}
}

```

Listing 14-2. Touch Events Module

```

module com.jfxbe.touchevents {
    requires javafx.controls;
    exports com.jfxbe.touchevents;
}

```

To build the module and run the project from command line with Java 9, run the steps from Listing 14-3.

Listing 14-3. Running the Touch Events Project

```

javac -d mods/com.jfxbe.touchevents $(find src/com.jfxbe.touchevents -name "*.java")
java --module-path build/modules -m com.jfxbe.touchevents/com.jfxbe.touchevents.TouchEvents

```

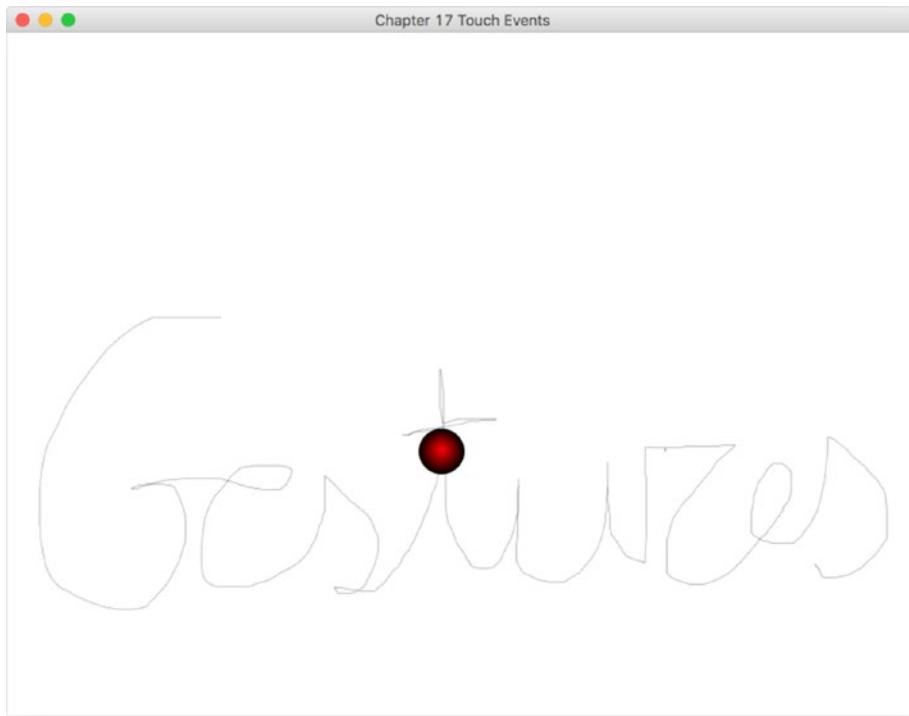


Figure 14-1. Using mouse or touch gestures to animate a shape along a path

How Does It Work?

This circle object is animated along the onePath path using a PathTransition with a 4000 millisecond transition time. The onePath point element list is defined using LineTo objects, which are built from the x and y coordinates captured from simple mouse event handlers. In this situation, the capture is initiated with the OnMousePressed event, collected by the OnMouseDragged event, and completed using the OnMouseReleased event.

However, as mentioned earlier, tracing is a very natural human interaction and is appropriately done with the hands. If you are running this on a touch-compliant screen device, you can easily use the JavaFX touch event support to enhance your tracing application. Touch events, like all gesture events, have first-class event handling in JavaFX 9. If a GUI component supports Mouse events, then it will also support gesture events. The best part is that Mouse and gesture events coexist nicely in a single application. Rather than replace the current Mouse event handlers, you can match each Mouse event handler with its equivalent Touch event handler. That is why the MousePressed, TouchPressed, MouseDragged, TouchMoved, MouseReleased, and TouchReleased events have been added to Listing 14-1.

As you can see, great care was taken to ensure that gesture event handling would be as seamless and natural as using traditional Mouse event handling.

Touching, Rotating, and Zooming in 3D

When you're manipulating a 3D scene, the most important interactions are typically rotations and zooming, which are most concerned with moving and changing the orientation of objects or the camera. Integrating gesture support into your 3D application is a natural enhancement over the traditional mouse and keyboard. JavaFX Shape3D objects, whether primitives or custom triangle meshes, support all gesture events, such as Rotate and Zoom, by default.

To demonstrate using gesture events within a JavaFX 3D scene, we add touchStationary, Rotate, and Zoom gesture events to a `TriangleMesh` example.

The base `TriangleMesh` example creates four pyramid `TriangleMesh` objects, each added to a `MeshView` container to be visible. Each `MeshView` container is added to a `Group` node, which is then added to the 3D scene graph.

We want to enhance the scene by adding a Rotate gesture event to each pyramid. A user should be able to touch a pyramid and then rotate it about the z-axis using a rotation gesture. If you read the previous example in this chapter, you know that we need to add an event handler for the Rotate gesture, in this case `OnRotate`. However, you must first provide a means to notify your application as to which onscreen object is intended to rotate, or else you should rotate the camera perspective. If you stay within the multi-touch paradigm, you will first need to add an event handler for `OnTouchStationary`. The `OnTouchStationary` event indicates to the application that a single touch has occurred and has remained in the same relative location. This differentiates the event from `OnTouchPressed` and `OnTouchMoved`, which occur immediately when any touch occurs on the device, whether it is part of a movement gesture or not.

The strategy is to select an onscreen object as the active node to be the focus of subsequent gestures. The `OnTouchStationary` event will occur before any `OnRotate` or `OnZoom` events, so the `activeNode` will be available to those following gesture events. You can use this event to select the active node by adding the following event handler:

```
scene.setOnTouchStationary(event -> {
    Node picked = event.getTouchPoint().getPickResult().getIntersectedNode();
    if (null != picked) {
        activeNode = picked;
    } else {
        activeNode = camera;
    }
    event.consume();
});
```

You can access the location data via the `TouchPoint` object. The `TouchPoint` object has picking support built-in, which automatically gives you access to the intersected node, which is to say the onscreen object `picked`. If no object is picked, that is, the touch was not over an onscreen object, then the camera is set as the `activeNode`. Now you can add the `OnRotate` event handler:

```
scene.setOnRotate(event -> {
    if (null == activeNode) {
        activeNode = camera;
    }
    activeNode.setRotationAxis(Rotate.Z_AXIS);
    activeNode.setRotate(activeNode.getRotate() + event.getAngle());
    event.consume();
});
```

Access the rotation value provided by the Rotate gesture using `event.getAngle()` and add it to whatever the previous rotation value was for the current node. This will spin the object onscreen. If the user is touching nothing, then the camera itself is rotated, which will cause the entire scene to appear to rotate. Figure 14-2 shows what your scene will look like when you apply a rotation gesture to several of the scene objects—in this case the filled ones.

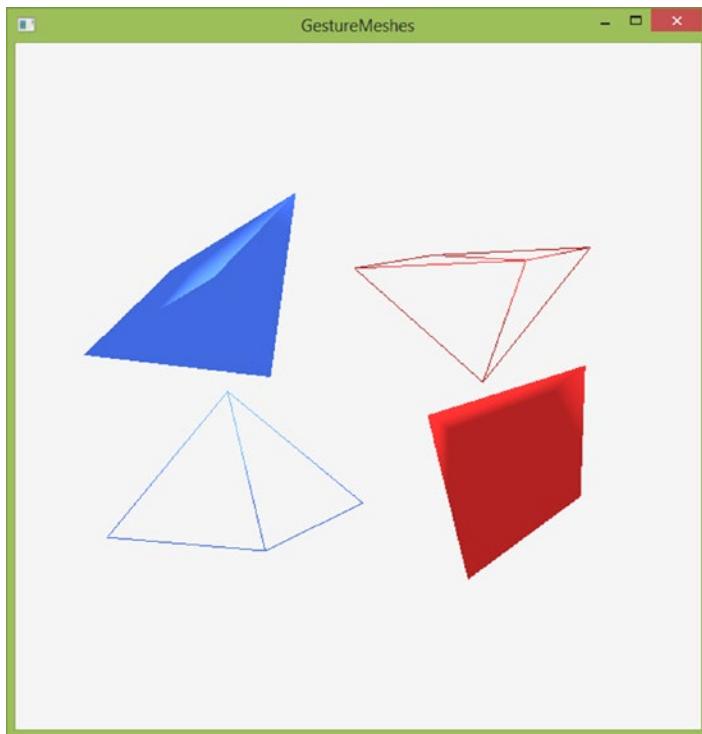


Figure 14-2. Rotating 3D pyramids using gestures

Note All objects in any scene are technically nodes. However, when picking objects, JavaFX returns a reference to the `MeshView` object. Any `Group` containers are ignored, as a `Group` container will present its contained node objects to the Picking API first, and the Grabbing API will only deliver the `TouchPoint` event to the topmost Node (in terms of z-order) onscreen.

Now let's follow the same pattern and give the user the ability to scale the `activeNode`. This will provide an effect of enlarging or shrinking the selected node. For this, you leverage the `OnZoom` event:

```
scene.setOnZoom(event -> {
    if(null == activeNode) {
        activeNode = camera;
    }
})
```

```

        double zoomFactor = event.getZoomFactor();
        zoomFactor *= zoomFactor < 1.0 ? -1 : 1;
        activeNode.setScaleX(activeNode.getScaleX() + zoomFactor / 50);
        activeNode.setScaleY(activeNode.getScaleY() + zoomFactor / 50);
        activeNode.setScaleZ(activeNode.getScaleZ() + zoomFactor / 50);
        event.consume();
    });
}

```

The `Zoom` event provides a double `ZoomFactor`, which is a positive value from 0.0 and up. Values returned between 0.0 and 1.0 indicate a “pinch” zoom gesture. Values returned higher than 1.0 indicate a “spread” zoom gesture. Here, you can use the “pinch” values to negatively scale the object while the “spread” values can positively scale the object. Figure 14-3 is an example of the screen when you apply a zoom gesture to several of the scene objects.

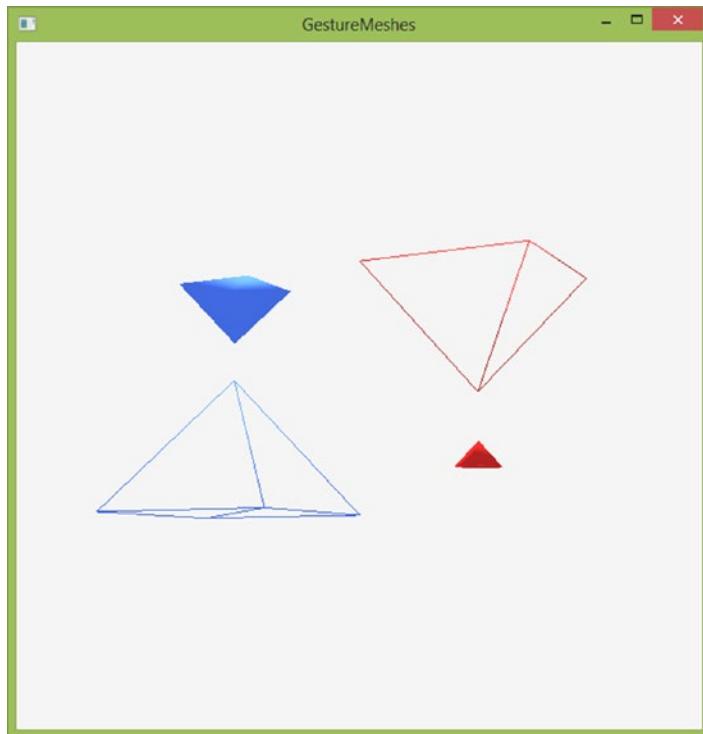


Figure 14-3. Zooming 3D pyramids using gestures

These three event handlers—`OnTouchStationary`, `OnRotate`, and `OnZoom`—have enabled multi-touch gesture support in your 3D scene. The effects stack on each other, as users can rotate and zoom simultaneously using a combination of “pinch” and “spin” movements with their fingers. Not only is this type of interaction quite natural to humans but is typically easier than with a mouse and keyboard solution.

Note Here, we are “scaling” our zoom with a fixed integer of 50. This is needed to compensate for the difference between onscreen coordinates and scaling coordinates. It is common practice when dealing with 3D scenes and can be thought of as a sensitivity variable.

The Leap Motion Controller

The Leap Motion controller (see Figure 14-4), also known as the Leap, is a small device, 80×30×12 mm, that is connected to your computer through the USB port. It doesn’t require an external power source.



Figure 14-4. Leap Motion device and its box

The Leap software works in Windows, MacOS X, and Linux, and with an initial price of \$79.99, was launched in July 2013 with an initial batch of 600,000 units sold around the world.

The company has some introductory videos here: <https://www.leapmotion.com/>. After you watch a few of them, you will be surprised by the apparently magic performance of the device.

Basically, the Leap will scan your hands at a very fast rate and translate your movements and gestures to the computer in a very precise way, so any Leap-enabled application can be interfaced directly, without the need of using a mouse, keyboard, or any other physical device.

As a new era of interaction with machines is beginning, many challenges are appearing at the same time. As users, we’ll need to learn how to interact in 3D environments, performing a whole new repertoire of gestures, while as developers, we will face the challenge of providing the software capable of translating accurately those gestures, so new applications will respond as we expect. With the hand tracking provided, Leap Motion is currently reaching into virtual and augmented reality to interact with new worlds, partnering with major VR manufacturers to embed Leap Motion technology into mobile VR/AR headsets.

In this chapter, we focus on the desktop controller and start by describing the hardware and software provided by Leap Motion. Then we will talk about the API for developing JavaFX programs with the device and present a sample application to show what is possible to achieve once you have a Leap device in your hands.

How It Works

Basically, the main hardware of the device consists of three infrared LEDs combined with two monochromatic infrared (IR) cameras. While the LEDs generate a 3D pattern of dots of IR light, the cameras scan the reflected data at nearly 300fps. Everything within a radius of 50cm will be scanned and processed. With a resolution of 0.01mm, data will be analyzed in the host computer by the Leap Motion software, using a proprietary motion detection algorithm.

Depending on your CPU capabilities and the data analyzed, the range of processing latency goes from 2ms to 33ms.

The best way to find out about the Leap magic is just plugging it on the USB port and installing the software from <https://www.leapmotion.com/setup>. A small Leap icon should appear on the taskbar notification area (Windows) or menu bar (Mac), and it should be green, like the lateral LED on the device (which should be facing you). Click on it and select the Visualizer for an incredible visualization of your fingers and hands.

When you see a grey grid, move your hands and click H to see the menu, N to visualize the hand, T the fingers, O to draw the gestures, or L to see the latency. You'll see something like Figure 14-5.

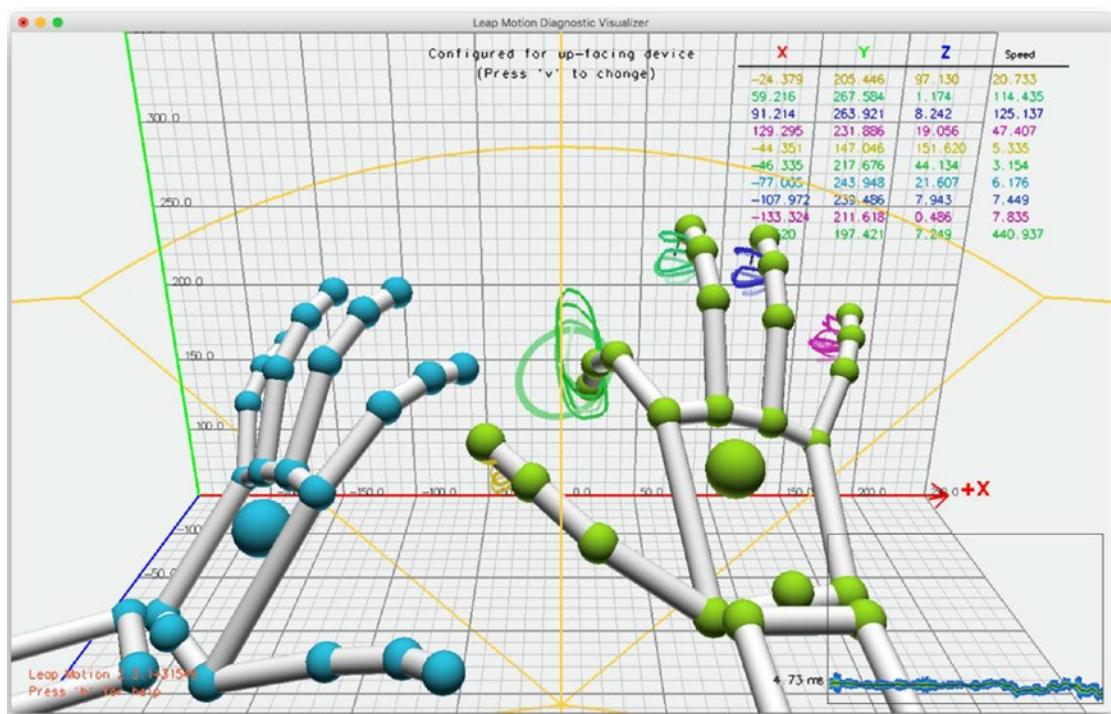


Figure 14-5. Leap Motion Diagnostic Visualizer application

As you can see, the device detects hands, fingers, and sharp tools within a 60cm range and a 120-150 view angle. For these objects, 3D vectors provide position, direction, and velocity, all referred to a right-handed Cartesian coordinate system, whose origin is at the center of the device, as shown in Figure 14-6. Every time the device scans and processes the available data, a Frame object is generated, containing lists with the tracked data in that instant (hands, fingers, and tools), as well as a set of basic motion gestures found in the analysis of the last frames (swipe, tap, and circles).

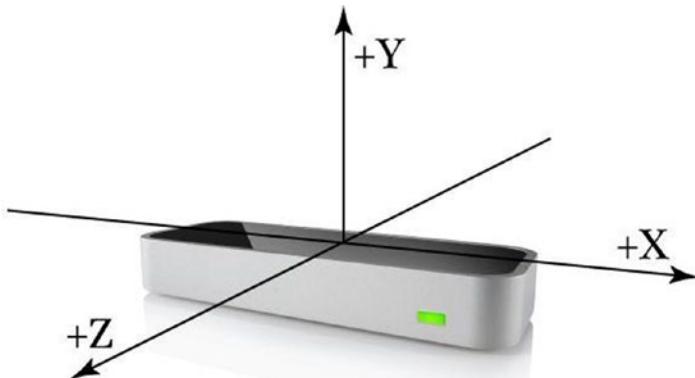


Figure 14-6. Coordinate system centered in the device

As you may have noticed, the y-axis positive direction is the opposite of the downward orientation in most computer graphics systems, including JavaFX. But the fact that the data is referred to the device position, and not to the screen as you are accustomed to with mouse and touch events, changes dramatically the way you need to think. Fortunately, the API provides several useful methods to find where your hands and fingers are pointing at any time.

More complex tasks require an understanding of spatial geometry, including knowledge of terms like vectors, vector-cross-products, and matrix operations.

Getting Started with the Leap SDK

In order to integrate the Leap device events and interact with software applications, Leap Motion provides an SDK for many programming languages, including Java, in their developer section. Make sure you select the V2 tracking version, intended for desktop only: <https://developer.leapmotion.com/sdk/v2>. Their new Orion beta version is only for Windows and it's mainly oriented to VR.

Download the SDK for your operating system, and you'll get the necessary libraries to get started. These consist of mainly the `LeapJava.jar` and a bunch of native libraries. The SDK also includes a `HelloWorld.java` example.

One easy way to integrate these files on your system is by adding the JAR to `<JAVA_HOME>/jre/lib/ext` on Linux or Windows (or `/Library/Java/Extensions` on Mac). Then copy the native libraries (`LeapJava.dll`, `Leap.dll` and `Leapd.dll` for Windows, `libLeapJava.dylib` and `libLeap.dylib` for Mac, `libLeapJava.so` and `libLeap.so` for Linux) to the `<JAVA_HOME>/bin` folder.

Or you can just add the JAR to every project as a dependency and load the native libraries as a VM argument: `-Djava.library.path=<native library path>`. This is what we'll do in the example, so you can see how this works with the modular system in Java 9.

Before going any further, look at the documentation at <https://developer.leapmotion.com/documentation/java/index.html> and try the `HelloWorld.java` example. You'll notice the following:

- You need a `Controller` object that allows the connection between the Leap device and the application.
- You need a `Listener` subclass to handle events from the controller.
- Gesture tracking is enabled in the `onConnect()` method.
- The main method in this class is `onFrame()`, which is a callback method dispatched when a new `Frame` with motion tracking data is available. Here, you can get lists of hands, fingers, or tools, as well as several vectors with their position, orientation, and velocity.
- If gestures are enabled, you'll also get a list of the gestures found, based on the analysis of the last frames. You will know as well the status of the gesture: whether it has just begun, is progressing, or has ended.

Adding the Leap SDK to a JavaFX Project

One thing you may have discovered throughout this book is the care you must put in mixing the so-called JavaFX thread with other non-JavaFX threads. At this point, it will be evident for you that the Leap Listener subclass runs in a non-JavaFX thread, handling events at a very high rate.

In order to bring these events to a JavaFX thread, you will use a JavaFX `ObjectProperty<T>` in the Listener subclass to store the desired values on every frame. Then in the JavaFX class, you'll implement a `ChangeListener<T>` to listen to any change and by using `Platform.runLater()`, you will put the task of rendering these changes on the JavaFX scene graph on a JavaFX thread.

The Hands Tracking Example

The following example uses the features of the Leap Motion v2 version and its skeletal tracking model to create a JavaFX 3D “live” version of the hand’s bones and arms.

Before getting started with the code, let’s set first the `module-info` class. Listing 14-4 shows the usual `requires` clause for JavaFX and the new one you add to include `LeapJava`.

Listing 14-4. HandsTracking Module

```
module com.jfxbe.handstracking {
    requires javafx.controls;
    requires LeapJava
    exports com.jfxbe.handstracking;
}
```

You must copy `LeapJava.jar` from the Leap SDK installation subfolder `LeapSDK/lib` to a `libs` folder within the project, and in the IDE, add this JAR to the libraries, in order to use its API, as shown in Figure 14-7.

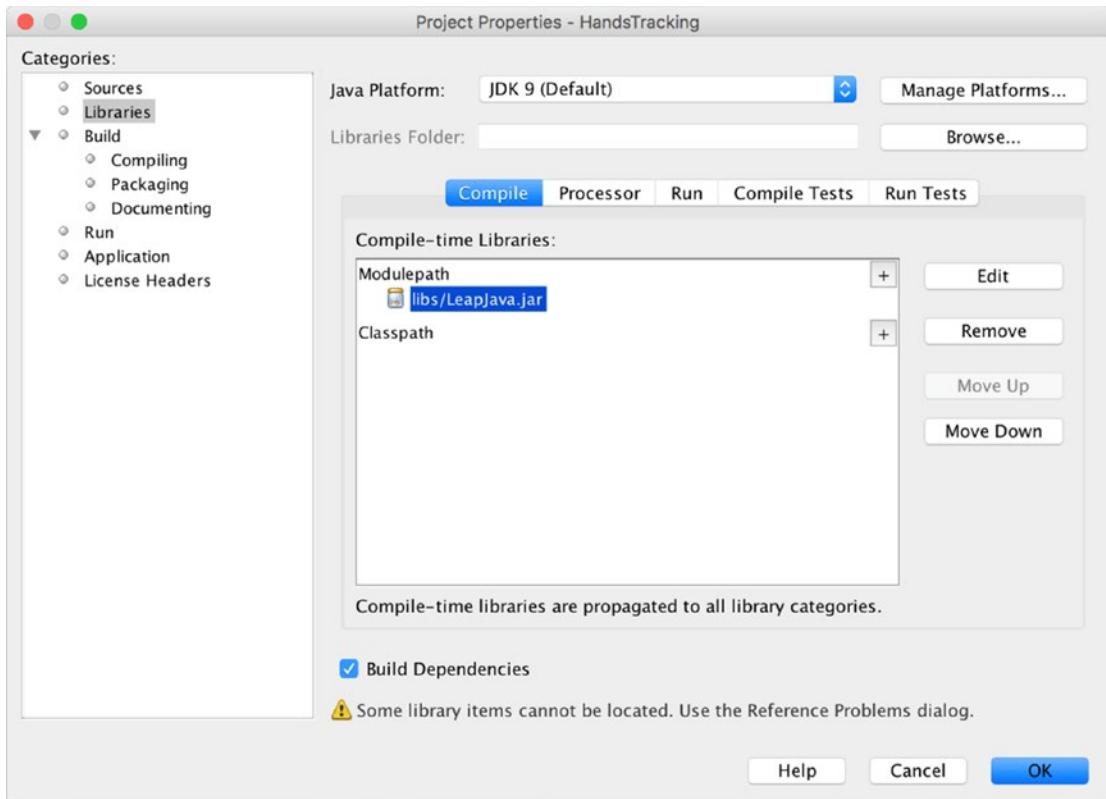


Figure 14-7. Adding the LeapJava.jar to the project libraries in NetBeans

The LeapListener Class

The LeapListener subclass will take care of tracking your hands, using the Leap Motion API, and for every frame it will return a list of valid bones, arms, and pair of joints. A BooleanProperty is used to mark the init and end of the hands processing, instead of having observable lists.

In Listing 14-5, the main method `onFrame` is called every few milliseconds if a valid controller is available (i.e., when you have the Leap Motion controller plugged in to your computer). On every frame, the API provides a list of fingers, and for each Finger, a list of Bones of a certain type.

Check the Leap tracking documentation for a detailed explanation of their API here:

https://developer.leapmotion.com/documentation/java/devguide/Leap_Overview.html.

Pair is an utility class that you use to track the joints and create a fake union between the metacarpophalangeal joints and the carpometacarpal joints, as show in Listing 14-6.

Listing 14-5. The LeapListener Class

```
/**
 * @author Jose Pereda
 */
public class LeapListener extends Listener {
```

```

private final BooleanProperty doneList= new SimpleBooleanProperty(false);
private final List<Bone> bones=new ArrayList<>();
private final List<Arm> arms=new ArrayList<>();
private final List<Pair> joints=new ArrayList<>();

@Override
public void onFrame(Controller controller) {
    Frame frame = controller.frame();
    doneList.set(false);
    bones.clear();
    arms.clear();
    joints.clear();

    if (!frame.hands().isEmpty()) {
        Screen screen = controller.locatedScreens().get(0);
        if (screen != null && screen.isValid()) {
            for (Finger finger : frame.fingers()) {
                if (finger.isValid()) {
                    for (Type b : Type.values()) {
                        if ((!finger.type().equals(Finger.Type.TYPE_RING) &&
                            !finger.type().equals(Finger.Type.TYPE_MIDDLE)) ||
                            !b.equals(Type.TYPE_METACARPAL)) {
                            bones.add(finger.bone(b));
                        }
                    }
                }
            }
            for (Hand h : frame.hands()) {
                if (h.isValid()) {
                    // arm
                    arms.add(h.arm());

                    FingerList fingers = h.fingers();
                    Finger index = null, middle = null, ring = null, pinky = null;
                    for (Finger f : fingers) {
                        if (f.isFinger() && f.isValid()) {
                            switch (f.type()) {
                                case TYPE_INDEX: index = f; break;
                                case TYPE_MIDDLE: middle = f; break;
                                case TYPE_RING: ring = f; break;
                                case TYPE_PINKY: pinky = f; break;
                            }
                        }
                    }
                    // joints
                    if (index != null && middle != null) {
                        Pair p = new Pair(index.bone(Type.TYPE_METACARPAL).nextJoint(),
                            middle.bone(Type.TYPE_METACARPAL).nextJoint(),
                            bones.get(0).width() / 2d);
                        joints.add(p);
                    }
                }
            }
        }
    }
}

```

```

        if (middle != null && ring != null) {
            Pair p = new Pair(middle.bone(Type.TYPE_METACARPAL).nextJoint(),
                               ring.bone(Type.TYPE_METACARPAL).nextJoint(),
                               bones.get(0).width() / 2d);
            joints.add(p);
        }
        if (ring != null && pinky != null) {
            Pair p = new Pair(ring.bone(Type.TYPE_METACARPAL).nextJoint(),
                               pinky.bone(Type.TYPE_METACARPAL).nextJoint(),
                               bones.get(0).width() / 2d);
            joints.add(p);
        }
        if (index != null && pinky != null) {
            Pair p = new Pair(index.bone(Type.TYPE_METACARPAL).prevJoint(),
                               pinky.bone(Type.TYPE_METACARPAL).prevJoint(),
                               bones.get(0).width() / 2d);
            joints.add(p);
        }
    }
}

doneList.set(!bones.isEmpty() || !arms.isEmpty());
}

public List<Bone> getBones() {
    return bones.stream().collect(Collectors.toList());
}
public List<Arm> getArms() {
    return arms.stream().collect(Collectors.toList());
}
public List<Pair> getJoints() {
    return joints.stream().collect(Collectors.toList());
}
public BooleanProperty doneListProperty() {
    return doneList;
}
}

```

Listing 14-6. The Pair Class

```

public class Pair {

    /*
    Creates a pair of joints (in terms of Vectors) to join the proximal end of two bones
    */
    private final Vector v0;
    private final Vector v1;
    private final double width;

```

```

public Pair(Vector v0, Vector v1, double width){
    this.v0 = v0;
    this.v1 = v1;
    this.width = width;
}
public Vector getCenter(){
    return new Vector((v1.getX() + v0.getX()) / 2f,
                      (v1.getY() + v0.getY()) / 2f,
                      (v1.getZ() + v0.getZ()) / 2f);
}
public Vector getDirection(){
    return new Vector(v1.getX() - v0.getX(),
                      v1.getY() - v0.getY(),
                      v1.getZ() - v0.getZ()).normalized();
}
// getters
}

```

The 3D Model Classes

The hand model will use JavaFX 3D Cylinder and Sphere controls to simulate its bones and joints. In order to render them in the proper location and orientation, you'll need to transform the directions you get from the Leap vectors into the JavaFX coordinate system. For that, you use the `Utils` class, shown in Listing 14-7.

Listing 14-7. The `Utils` Class

```

public class Utils {

    public static List<Transform> getTransforms(Vector dir, Vector pos, double yOffset) {
        final double ang = Math.acos(dir.getY() / dir.magnitude());
        return Arrays.asList(
            new Translate(pos.getX(), -pos.getY() + yOffset, -pos.getZ()),
            new Rotate(-Math.toDegrees(ang), 0, -yOffset, 0,
                      new Point3D(-dir.getZ(), 0, -dir.getX())));
    }
}

```

`getTransforms` returns a list of transforms based on a direction and a position. It takes as input the vectors from the Leap Motion coordinate system and transforms them into a translation and a rotation over the JavaFX coordinate system.

A vertical cylinder is taken as a reference for the transformations, located at the given position. The translation is given by the item position in the JavaFX system (`pos.getX()`, `-pos.getY()`, `-pos.getZ()`). The pivot point for the rotation is given by the cross product of the item direction in the JavaFX system (`dir.getX()`, `-dir.getY()`, `-dir.getZ()`) and the cylinder direction (`0`, `-1`, `0`) in the same system. The angle of rotation is given by the angle between both referred vectors. An offset can be applied if required, from the center of the cylinder.

Listings 14-8, 14-9, and 14-10 show the 3D model for phalanx, joints, and forearm.

Listing 14-8. The Phalanx Class

```

public class Phalanx {

    private final Bone bone;
    private final Cylinder phalanx;
    private final Sphere joint;
    private final Sphere carpoMetaCarpalJoint;

    public Phalanx(Bone bone) {
        this.bone = bone;
        PhongMaterial materialPhalanx = new PhongMaterial();
        materialPhalanx.setSpecularColor(Color.rgb(50, 50, 50));
        PhongMaterial materialJoint = new PhongMaterial(Color.BLUE);
        materialJoint.setSpecularColor(Color.rgb(50, 50, 50));

        phalanx = new Cylinder(0.8 * bone.width() / 2d, bone.length());
        phalanx.setMaterial(materialPhalanx);
        phalanx.getTransforms().addAll(
            Utils.getTransforms(bone.direction(), bone.center(), 0));

        joint = new Sphere(bone.width() / 2d);
        joint.setMaterial(materialJoint);
        joint.getTransforms().addAll(
            Utils.getTransforms(bone.direction(), bone.center(), bone.length() / 2d));

        carpoMetaCarpalJoint = new Sphere(bone.width() / 2d);
        carpoMetaCarpalJoint.setMaterial(materialJoint);
        carpoMetaCarpalJoint.getTransforms().addAll(
            Utils.getTransforms(bone.direction(), bone.center(), -bone.length() / 2d));
    }

    public Group getPhalanx() {
        if (bone.type().equals(Type.TYPE_METACARPAL)) {
            return new Group(phalanx, carpoMetaCarpalJoint);
        } else if (bone.type().equals(Type.TYPE_PROXIMAL)) {
            return new Group(phalanx, joint, carpoMetaCarpalJoint);
        }
        return new Group(phalanx, joint);
    }
}

```

Listing 14-9. The Joint Class

```

public class Joint{
    private final Cylinder cylinder;

    public Joint(Pair pair) {
        PhongMaterial material = new PhongMaterial();
        material.setSpecularColor(Color.rgb(30, 30, 30));

```

```

        cylinder = new Cylinder(pair.getWidth() / 2d, pair.getV0().distanceTo(pair.getV1())));
        cylinder.setMaterial(material);
        cylinder.getTransforms().addAll(
            Utils.getTransforms(pair.getDirection(), pair.getCenter(), 0));
    }

    public Shape3D getJoint() {
        return cylinder;
    }

}

```

Listing 14-10. The ForeArm Class

```

public class ForeArm{
    private final Cylinder cylinder;

    public Forearm(Arm arm) {
        PhongMaterial materialArm = new PhongMaterial();
        materialArm.setDiffuseColor(Color.CORN SILK);
        materialArm.setSpecularColor(Color.rgb(30, 30, 30));

        cylinder = new Cylinder(arm.width() / 2d,
            arm.elbowPosition().minus(arm.wristPosition()).magnitude());
        cylinder.setMaterial(materialArm);
        cylinder.getTransforms().addAll(
            Utils.getTransforms(arm.direction(), arm.center(), 0));
    }

    public Shape3D getForearm() {
        return cylinder;
    }

}

```

The Application Class

Finally, `HandsTracking`, which is shown in Listing 14-11, is the JavaFX Application that will create a new `Controller` instance that will start listening to frames based on a `LeapListener` instance.

For every frame, `listener.doneListProperty()` will change to true when the frame is processed, so at this point you'll process the lists of (Leap) bones, pairs, and arms to create and render the (JavaFX 3D) phalanges, joints, and forearms. Note that a `Group` holding the collection of these 3D shapes is filled using `Platform.runLater()`. This means that if the frame rate is too high, not all the frames will be processed, but there won't be concurrent modification exceptions when the lists change from one frame to the other, as you are getting copies of the list being modified.

Listing 14-11. The Utils Class

```
public class HandsTracking extends Application {

    private LeapListener listener = null;
    private Controller controller = null;

    @Override
    public void start(Stage primaryStage) throws Exception {
        listener = new LeapListener();
        controller = new Controller();
        controller.addListener(listener);

        final PerspectiveCamera camera = new PerspectiveCamera();
        camera.setFieldOfView(60);
        camera.getTransforms().addAll(new Translate(-400, -500, 200));

        final PointLight pointLight = new PointLight(Color.ANTIQUEWHITE);
        pointLight.setTranslateX(0);
        pointLight.setTranslateY(-1000);
        pointLight.setTranslateZ(-800);

        final Group root = new Group(pointLight);

        final Group root3D = new Group(camera, root);
        final SubScene subScene = new SubScene(root3D, 800, 600, true, SceneAntialiasing.BALANCED);
        subScene.setCamera(camera);

        Scene scene = new Scene(new AnchorPane(subScene), 800, 600, Color.WHITESMOKE);

        listener.doneListProperty().addListener((obs, ov, nv) -> {
            if (nv) {
                List<Bone> bones = listener.getBones();
                List<Arm> arms = listener.getArms();
                List<Pair> joints = listener.getJoints();

                Platform.runLater(() -> {
                    root.getChildren().setAll(pointLight);

                    root.getChildren().addAll(bones.stream()
                        .filter(bone -> bone.isValid() && bone.length() > 0)
                        .map(Phalanx::new)
                        .map(Phalanx::getPhalanx)
                        .collect(Collectors.toList()));

                    root.getChildren().addAll(joints.stream()
                        .map(Joint::new)
                        .map(Joint::getJoint)
                        .collect(Collectors.toList())));
                });
            }
        });
    }
}
```

```

        root.getChildren().addAll(arms.stream()
            .filter(Arm::isValid)
            .map(Forearm::new)
            .map(Forearm::getForearm)
            .collect(Collectors.toList())));
    });
}
});

primaryStage.setTitle("Hand Tracking with Leap Motion");
primaryStage.setScene(scene);
primaryStage.show();
}

@Override
public void stop(){
    controller.removeListener(listener);
}
}
}

```

Building and Running the Project

To build the module and run the project from the command line with Java 9, run the steps from Listing 14-12. Be aware that in order to build the module, you need to include the `LeapJava.jar` with the `module-path` option, and in order to run the project, you need to add the native libraries with `-Djava.library.path`.

Listing 14-12. Running the Hands Tracking Project

```

javac --module-path libs/LeapJava.jar -d mods/com.jfxbe.handstracking
$(find src/com.jfxbe.handstracking -name "*.java")

java -Djava.library.path=<path to>/LeapSDK/lib/ --module-path mods:libs -m com.jfxbe.
handstracking/com.jfxbe.handstracking.HandsTracking

```

If everything is in place and you have your Leap Motion Controller plugged in, you should be able to wave your hand on top of it and get something like Figure 14-8.

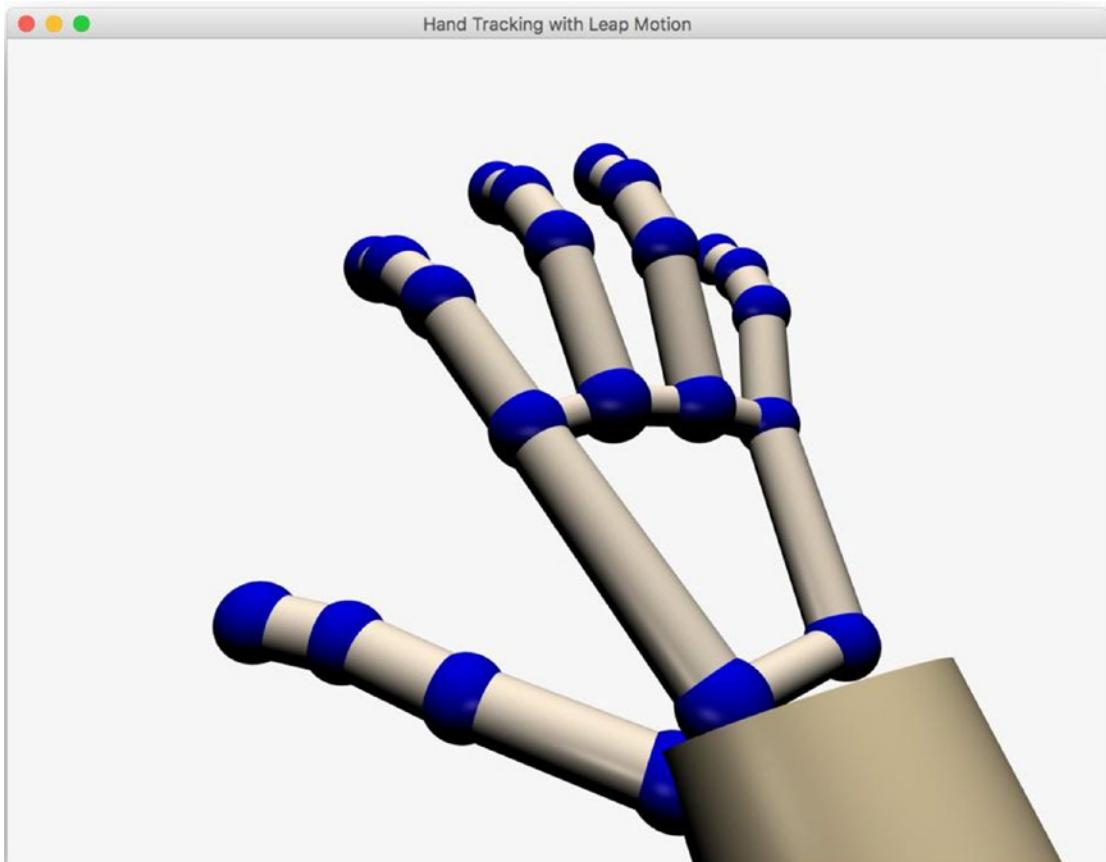


Figure 14-8. Running the hands tracking example

You can check with the Visualize application that the JavaFX 3D model you have built is almost the same as the native one, as shown in Figure 14-9.

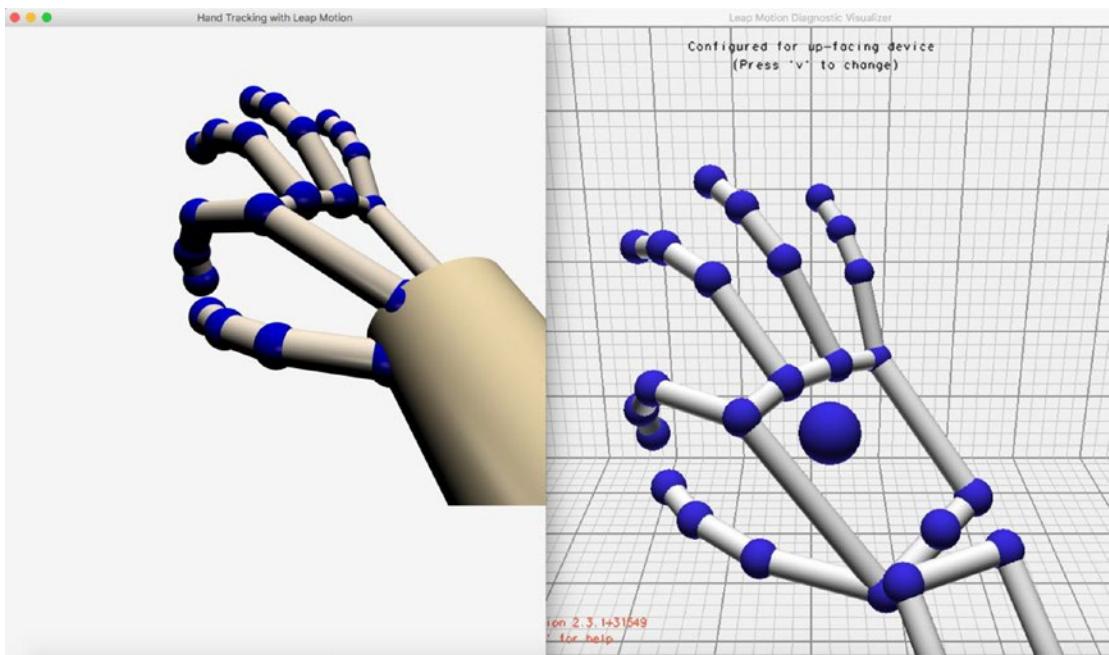


Figure 14-9. Comparing the hands tracking model with visualization

More Examples

For more examples of using JavaFX with the Leap Motion device, refer to online resources such as:

- <http://jperedadnr.blogspot.com.es/2013/06/leap-motion-controller-and-javafx-new.html>
- <http://jperedadnr.blogspot.com.es/2015/01/creating-and-texturing-javafx-3d-shapes.html#SkinningMeshes>

For interaction with other programming languages, go to <https://gallery.leapmotion.com>.

Summary

In the first part of this chapter, you were introduced to various gesture events and the subtle differences between them. The events you learned included `TouchPressed`, `TouchReleased`, `TouchStationary`, `Rotate`, and `Zoom`. You learned how to add gesture event handling to your JavaFX application, whether it was a traditional 2D or even a 3D scene.

In the second part of this chapter, you discovered the impressive Leap Motion device, and the really nice combination effects that result from using it to enhance JavaFX applications. You began by learning about the device and how it works; next, you saw briefly its SDK for Java, along with a sample application, where you learned about listening and processing the data from the Leap device in one thread, while rendering 3D shapes in the JavaFX thread that track your own hands.

In the next chapter, you will learn about theming and how to customize applications, about the fundamentals of JavaFX CSS styling, and about creating custom controls.

CHAPTER 15



Custom UIs

Wouldn't it be nice if you could change the appearance of your application without altering its functionality? Many companies spend lots of money and time designing user interfaces (UIs).

Often companies will mock up several designs before deciding how their UI should look and behave. Depending on the scope, many companies also build UIs that support different screen dimensions. Typically, tablets, smartphones, TVs, and desktops are among the main challenges UI designers face.

In this chapter, you learn about theming and how to customize applications by applying various themes (look and feels). Next, you learn about the fundamentals of JavaFX CSS styling, which will enable you to change attributes of any Scene graph node, such as colors, backgrounds, margins, and so on. After learning about CSS styling, you learn about creating custom controls. Here, you learn how to build UI controls that aren't part of the JavaFX 8 built-in controls. Some custom controls that might come to mind are LEDs, number pads, gauges, clocks, and futuristic context menus.

In this chapter, you learn about the following APIs and concepts:

- Theming
- JavaFX CSS styling
- Custom controls

Theming

You have probably heard the terms *skinning* and *look and feel* in the context of designing user interfaces. These terms are often used interchangeably, and they both reflect the basic concept of theming or styling a UI. Theming is the idea of transforming an entire application's appearance without changing its underlying functionality. An example of theming is when you change an application's UI layout and controls from a stylized business theme to a sports-like theme. Usually, companies try to skin an application to resemble their logo. They also use beautiful typefaces (typography) selected to personify their corporate culture or target audience.

In JavaFX you have the ability to create, modify, or use existing themes to skin your applications. Later, I discuss how to create themes, but for now I show you how to apply existing themes (skins) in JavaFX applications. To demonstrate applying themes to an application's UI, I created a JavaFX application depicted in Figure 15-1, which allows users to apply various UI look-and-feel themes.

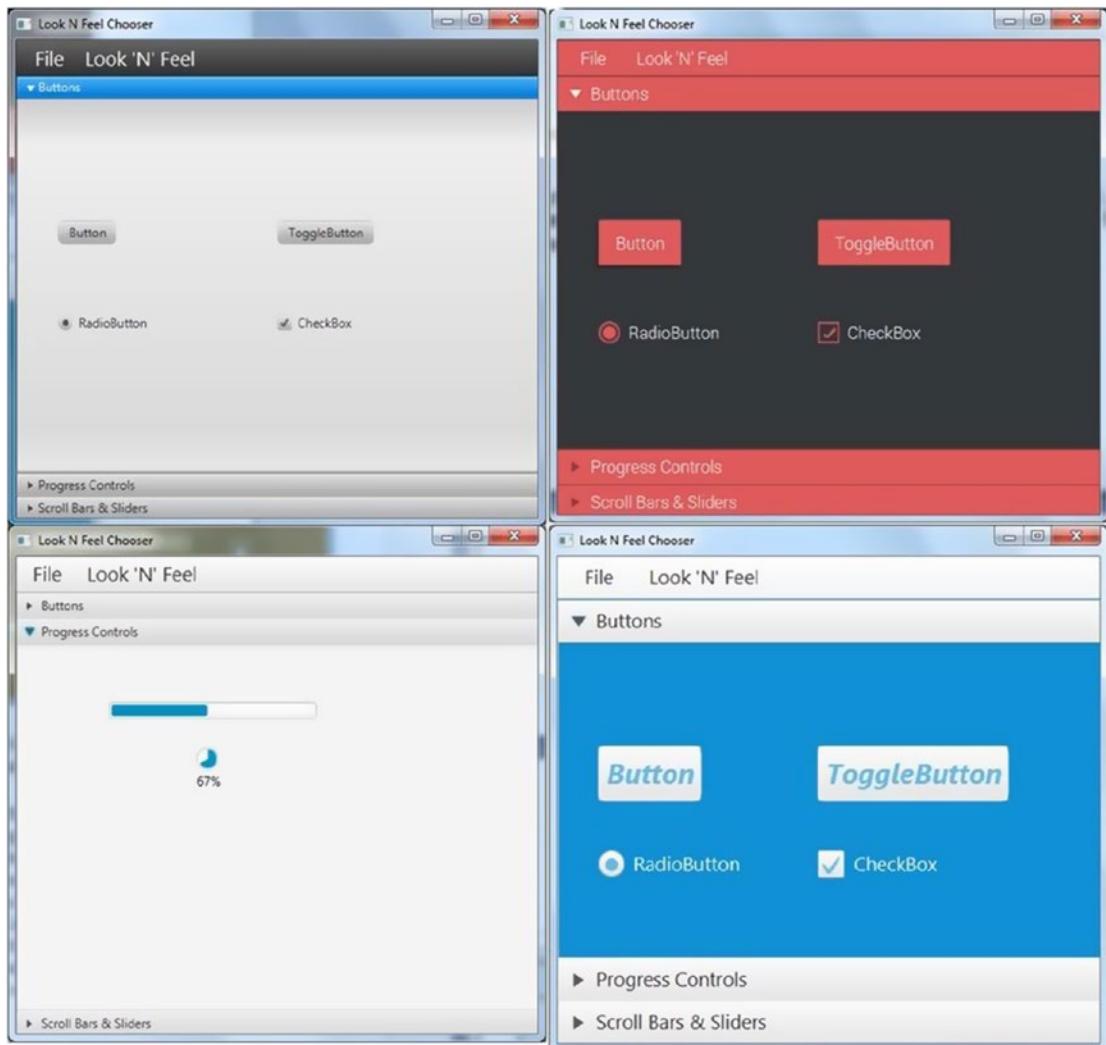


Figure 15-1. Setting an application's look and feel using JavaFX CSS styling. Moving counterclockwise from the upper-left are the Caspian, Modena, Sky, and Flat Red themes.

As you can see, there are four UI skins applied to the same application. At the upper left is the familiar Caspian theme from Java FX 2.X. Caspian has served well over the years and still has quite the professional look. At the lower left you see an accordion UI control expanded to show progress controls using JavaFX 8's new default theme, *Modena*. The Modena theme seems brighter, cleaner, and more modern.

At the lower right, the next theme, called the *Sky* look and feel, was created by yours truly (Carl Dea). My intent was to create a theme for touch displays so that the controls would be slightly enlarged for adult-sized fingers. I wanted a friendly feel and so the cool blue sky came to mind.

Last, but not least, in the upper right is the *Flat Red* look and feel, created by Gerrit Grunwald.

The *Flat Red* uses a custom font (Roboto) to give text on controls a crisp and clean look. Another nice feature of the *Flat Red* theme is that it uses CSS effects based on pseudo-class state changes of UI controls, such as a mouse press on a slider control. Don't worry if you don't know yet what pseudo-classes are, because later in this chapter, you will learn about pseudo-class selectors in detail. Mr. Grunwald's look and feel is also an excellent candidate for touch displays.

Native Look and Feels

Before jumping into example code, I want to give a shout-out to individuals (pioneers) in the JavaFX community who have crafted beautiful skins and themes that you will find most inspiring. For those interested in the OS X (Mac desktop) native look and feel, Claudine Zillmann (@etteClaudette) created AquaFX (with Elements). You can download AquaFX at <http://aquafx-project.com>. AquaFX, shown in Figure 15-2, is a library that allows developers to style applications easily with the native OS X look, but also to theme the UI with colors other than the usual aqua blue.

The screenshot shows the AquaFX website's 'Styled Controls' page. On the left, there's a sidebar with links like 'Introduction', 'Quickstart', 'Using Aqua', 'Architecture', 'Control Variations', and 'Styled Controls'. The 'Styled Controls' link is highlighted. The main content area has a title 'Styled Controls' and a subtitle 'A listing of all controls, which are styled by AquaFX.' Below this is a table with columns 'Component', 'Preview', and 'StylerClass'. The table lists various JavaFX controls and their corresponding styled classes:

Component	Preview	StylerClass
Label	Label	LabelStyler
Hyperlink	hyperlink	-
Button	Button	ButtonStyler
ToggleButton	ToggleButton	ToggleButtonStyler
CheckBox	<input type="checkbox"/> CheckBox <input checked="" type="checkbox"/> CheckBox <input type="checkbox"/> CheckBox	CheckBoxStyler
RadioButton	<input type="radio"/> RadioButton <input checked="" type="radio"/> RadioButton	RadioButtonStyler
ChoiceBox	A ▾	ChoiceBoxStyler
ComboBox	select ▾ select ▾	ComboBoxStyler
ColorPicker	<input type="color" value="#cce6ff"/> cce6ff ▾	ColorPickerStyler
MenuButton	Shutdown ▾	-
SplitMenuButton	Shutdown ▾	-
Slider		SliderStyler

Figure 15-2. AquaFX is a JavaFX library that styles controls resembling the OS X native look and feel

For those interested in Window's Metro look and feel, Pedro Duque Vieira (@P_Duke) created a project called JMetro. JMetro has a light theme, shown in Figure 15-3, and a dark theme shown in Figure 15-4.

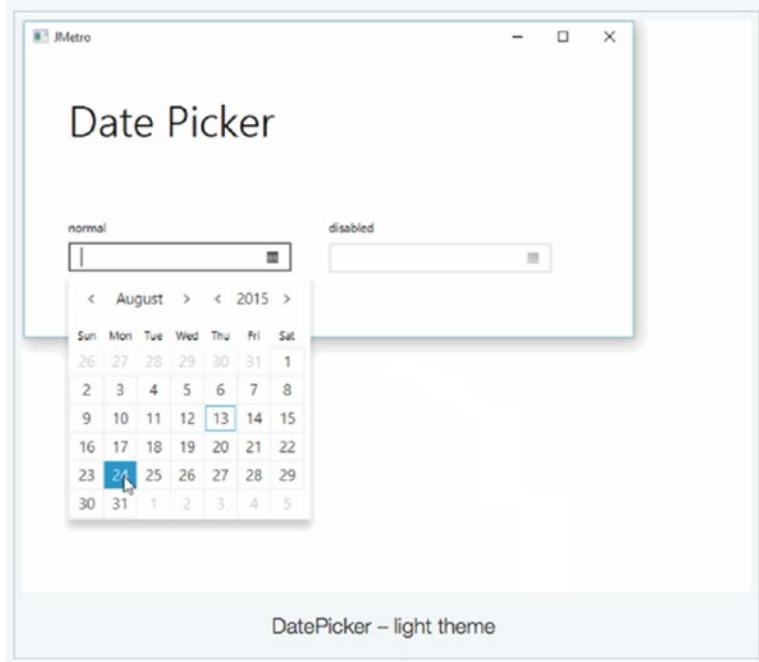


Figure 15-3. The date picker control stylized using JMetro's light theme

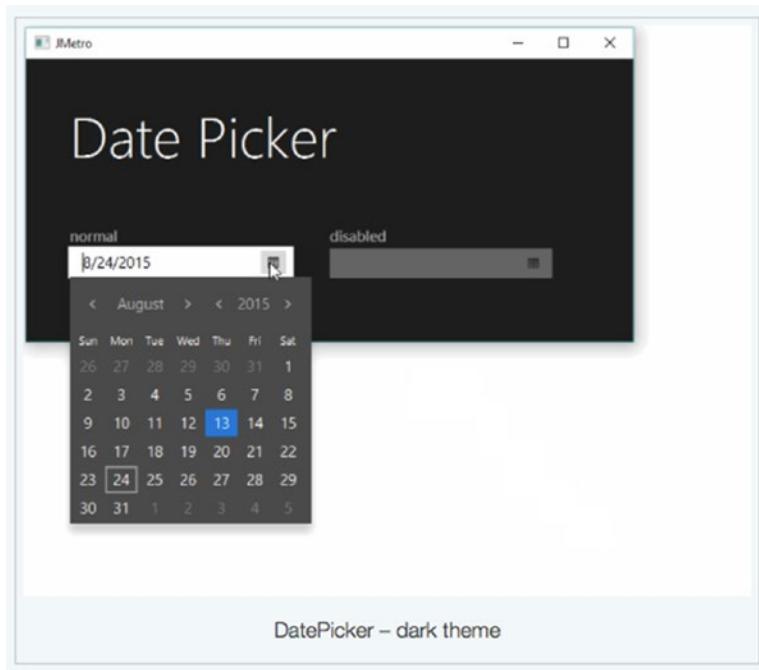


Figure 15-4. The date picker control stylized using JMetro's dark theme

To download the JMetro theme, go to JFXtras' Style section at <https://github.com/JFXtras/jfxtras-styles/tree/master/src>.

What about a look and feel that has a Roku-like or Apple TV-like interface; it's called Flatter, by Java Champion and book author Hendrik Ebbers (@hendrikEbbers). Flatter was designed for a proof of concept project BoxFX. BoxFX is another project Mr. Ebbers created by using a Raspberry Pi device running JavaFX 8 with the Flatter look and feel, as shown in Figure 15-5.

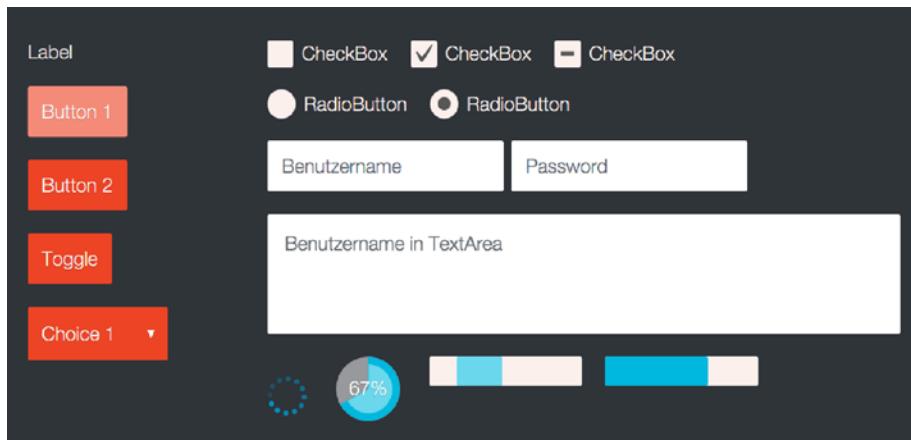


Figure 15-5. The Flatter style applied to a JavaFX application

The following are links to the UI styles mentioned in this section:

- AquaFX (with Elements): <http://aquafx-project.com/project.html>
- Jmetro
 - <http://pixelduke.wordpress.com/category/metro>
 - <https://github.com/JFXtras/jfxtras-styles>
- Flatter: <http://www.guigarage.com/2013/09/flatter>
- BoxFX: <http://www.guigarage.com/2013/08/boxfx-javaone-preview-1>

Now that you've seen native looking styles and media device styles, let's look at web and mobile UI styles.

Web and Mobile Look and Feels

Depending on the application, the users might prefer a more web or mobile look and feel. Often the screen dimensions of a device can force a designer's decision to have a non-native UIs look. Non-native UIs make up the majority of popular web sites and mobile apps. So what are the popular non-native UIs that I'm referring to?

As a web developer, the most popular UI styles (look and feels) today are Google's *Material Design* and Twitter's *Bootstrap*. For example, Figure 15-6 shows a JavaFX clock and TreeTableView controls skinned with the Material Design style.

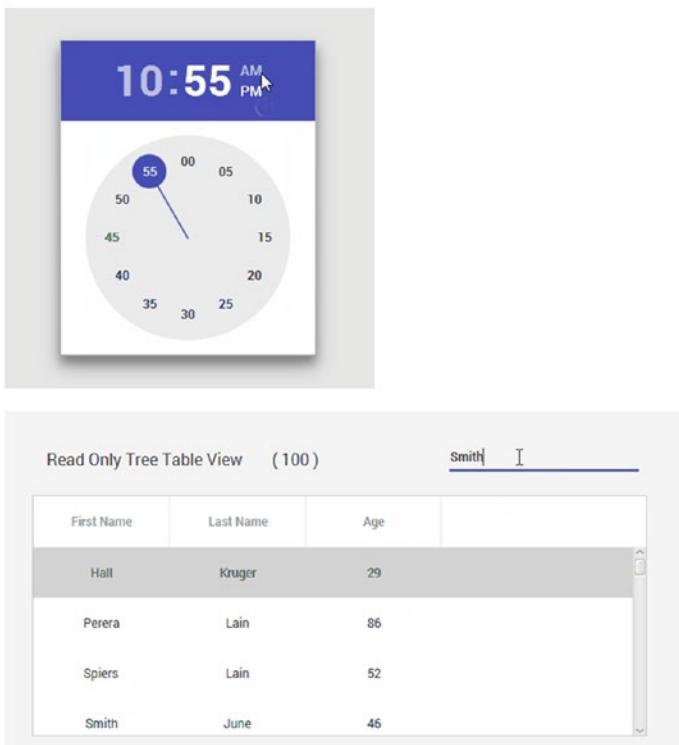


Figure 15-6. The Material Design style applied to a JavaFX application

Figure 15-7 shows buttons styled with the Twitter's *Bootstrap* look and feel.

Buttons



Figure 15-7. Twitter's *Bootstrap* style applied to a button controls

As the JavaFX community continues to grow and adapt to these new UI styles, there are a growing number of developers who have ported these popular UI styles as libraries to help you transform your JavaFX application to these popular UI styles.

New to the list of folks who have created popular web and mobile device themes are the following:

- (Google) Material Design
 - GluonHQ
 - Site: <http://gluonhq.com/products/mobile/>
 - JFoenix
 - Site: <http://www.jfoenix.com>
 - Author(s): <http://www.jfoenix.com/team.html>
 - MaterialFX
 - Site: <http://www.agix.pt/single-post/2015/09/02/MaterialFX-Material-Design-CSS-for-JavaFX>
- (Twitter) Bootstrap
 - JBootx
 - Site: <https://github.com/dicolar/jbootx>
 - Author(s): <https://github.com/dicolar/jbootx/graphs/contributors>
 - BootstrapFX
 - Site: <https://github.com/aalmiray/bootstrapfx>
 - Author: Andres Almiray

Applying the JavaFX CSS Theme

Now that you have seen a glimpse of some great-looking skins and UI styles, let's see how to apply JavaFX CSS-based styles. JavaFX has the capability to apply cascading style sheets to the Scene graph and its nodes in very much the way web developers use CSS style sheets with HTML5 elements. These CSS style sheets are external files containing attributes and values to style JavaFX nodes. An example JavaFX CSS file looks something like Listing 15-1.

Listing 15-1. A JavaFX CSS File Containing Styling Attributes

```
/* sample.css */
.button {
    -fx-text-fill: brighter-sky-blue;
    -fx-border-color: rgba(255, 255, 255, .80);
    -fx-border-radius: 8;
    -fx-padding: 6 6 6 6;
    -fx-font: bold italic 20pt "LucidaBrightDemiBold";
}
/* ...more styling */
```

This code contains a CSS class style definition for JavaFX button controls. Button UI controls by default have a class style button, which maps to -fx- prefixed properties. You will learn about JavaFX CSS styling later in this chapter, but for now let's discuss how to swap between the CSS style sheets that will dynamically skin an application.

In this section, you learn the two ways to apply CSS style sheets as look-and-feel themes onto JavaFX applications. The following are the two ways to apply JavaFX CSS files to Scene graph nodes.

- `javafx.application.Application: setUserAgentStylesheet()`
- `javafx.scene.Scene: getStylesheets().add()`

Using the `setUserAgentStylesheet(String URL)` Method

The first way to apply CSS style sheets is by invoking the static `setUserAgentStylesheet()` method on the JavaFX Application (`javafx.application.Application`) class. This static method styles every scene and all child nodes in a JavaFX application as shown here.

```
Application.setUserAgentStylesheet(url);
```

The `setUserAgentStylesheet(String URL)` method accepts a valid URL string representing the JavaFX CSS file. Typically, CSS files are bundled inside a JAR application; however they can reside on the local filesystem or a remote web server. When the CSS file is in the classpath, the call to the following method will find the CSS file and produce an URL string for accessing the file:

```
getClass().getResource("path/some_file.css").toExternalForm()
```

This assumes the resource is copied and collocated where the compiled classes are.

The following code snippet loads the `sample.css` file as the current look and feel for the JavaFX Application. The `sample.css` file in the code snippet is co-located where the current class is located. In other words, your Java class and your CSS file are in the same directory, so there is no need for a path in front of the filename.

```
Application.setUserAgentStylesheet(getClass().getResource("sample.css")
    .toExternalForm());
```

The `setUserAgentStylesheet()` method applies styling globally to all scenes owned by an application, such as context menus and child popup windows. JavaFX 8 currently contains two style sheets—*Caspian* and *Modena*—which serve as default cross-platform look and feel skins. Because the two style sheets are predefined, you can easily switch between them using the `setUserAgentStylesheet()` method. The following code shows how to switch between the Caspian and Modena look and feel style sheets.

```
// Switch to JavaFX 2.x's CASPIAN Look and Feel.
Application.setUserAgentStylesheet(STYLESHEET_CASPIAN);

// Switch to JavaFX 8's Modena Look and Feel.
Application.setUserAgentStylesheet(STYLESHEET_MODENA);
```

A great way to learn how the pros do it (creating skins) is to look at good examples. You can extract the CSS files (`caspian.css` and `modena.css`) from the `jfxrt.jar` file or view the JavaFX source code, located at <http://openjdk.java.net>. To obtain `jfxrt.jar`, you must be on the JDK prior to Java 9.

Since Java 9 no longer makes the `jfxrt.jar` available on the runtime or JDKs, you need to look for the `jmods` directory under `JAVA_HOME`. Part of project Jigsaw was also restructuring the runtime as modules. In the `jmods` directory you should find the module called `javafx.controls.jmod`. In order to get into the module file, you have to copy the module to some other temporary directory. Next, you should rename the copied module and end it with a `.jar` extension to be extracted. Lastly, you invoke `jar xvf` on the renamed module. Use your file explorer or Finder to locate the following file:

```
classes/com/sun/javafx/scene/control/skin/modena/modena.css
```

The `modena.css` file contains all the JavaFX CSS styling for the default look and feel. This file is where you can learn from the experts as to how they style every control on the JavaFX platform.

Note When you invoke the `setUserAgentStylesheet(null)` method by passing in a null value, the default style sheet (Modena) will be automatically loaded and set as the current look and feel. However, if you are using JavaFX 2.x, the default style sheet will be Caspian.

Using Scene's `getStylesheets().add(String URL)` Method

The second way of applying a look and feel is by invoking a `Scene` object's `getStylesheets().add()` method. Unlike the first method of applying a look and feel, the `getStylesheets().add()` method is used to style a single `scene` and its child nodes.

By calling the `getStylesheets().add()` method, you can style a single `scene` and its child nodes as shown:

```
Scene scene = ...
scene.getStylesheets().add(getClass().getResource("sample.css")
    .toExternalForm());
```

As in the previous call to `setUserAgentStylesheet()`, you pass in an URL string that represents a JavaFX CSS file. Later, you will see an example application that switches between different styles by using either the `setUserAgentStylesheet()` or the `getStylesheets().add()` method. If you decide to create an entire CSS to style an entire app, you maybe hesitant to start from scratch.

Creating an entire look and feel requires hundreds if not thousands of lines of code in order to style every UI control in JavaFX. So, it's a good idea to start off with the default look and feel and then override styles using the `getStylesheets().add()` method.

I basically created very small look-and-feel style sheet files to demonstrate the ability to swap themes. The simple example CSS files have just a handful of CSS styles for a small amount of UI controls that would be styled. In other words, the example's small CSS files are loaded via the `getStylesheets().add()` method, which styles only some of the nodes and not every node in the application.

Because the default style sheet (`Modena.css`) is loaded from the prior call via the `setUserAgentStylesheet(null)` method, the smaller custom CSS styling file can then “piggyback” onto the default style sheet. What's nice is that you don't have to start from scratch to create a new look and feel. The following snippet of code initially invokes the `setUserAgentStylesheet()` method with a null value, which loads Modena as the default look and feel. It then sets the scene's additional styling by invoking the `getStylesheets().add()` method.

```
Application.setUserAgentStylesheet(null); // defaults to Modena  
  
// apply custom look and feel to the scene.  
scene.getStylesheets()  
    .add(getClass().getResource("my_cool_skin.css")  
        .toExternalForm());
```

An Example of Switching Themes

To show you how to switch between various CSS style sheets (aka skins or themes), I've created an example application called the *Look N Feel Chooser*, which allows you to choose between different predefined themes. Figure 15-8 shows the Look and Feel Chooser example application initially displaying an accordion UI control pane with common UI controls, beginning with the Modena look and feel.

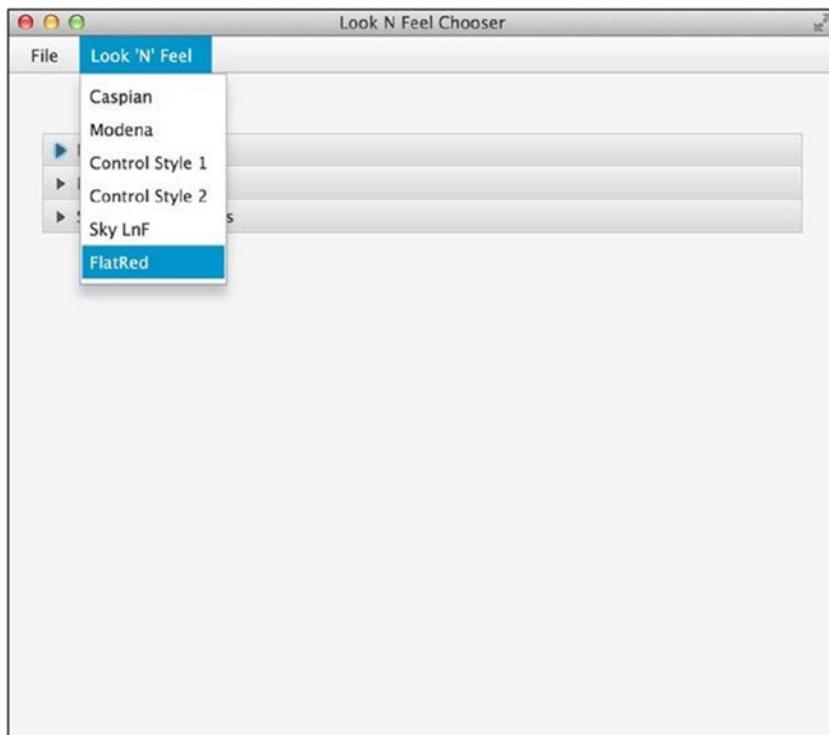


Figure 15-8. Predefined look and feel themes to choose from the Look N Feel menu

In Figure 15-8, you'll notice the menu options to select a look and feel (skin). Once a look and feel has been selected, the application's appearance will change dynamically. Figure 15-9 shows the application switched to the Flat Red look and feel.

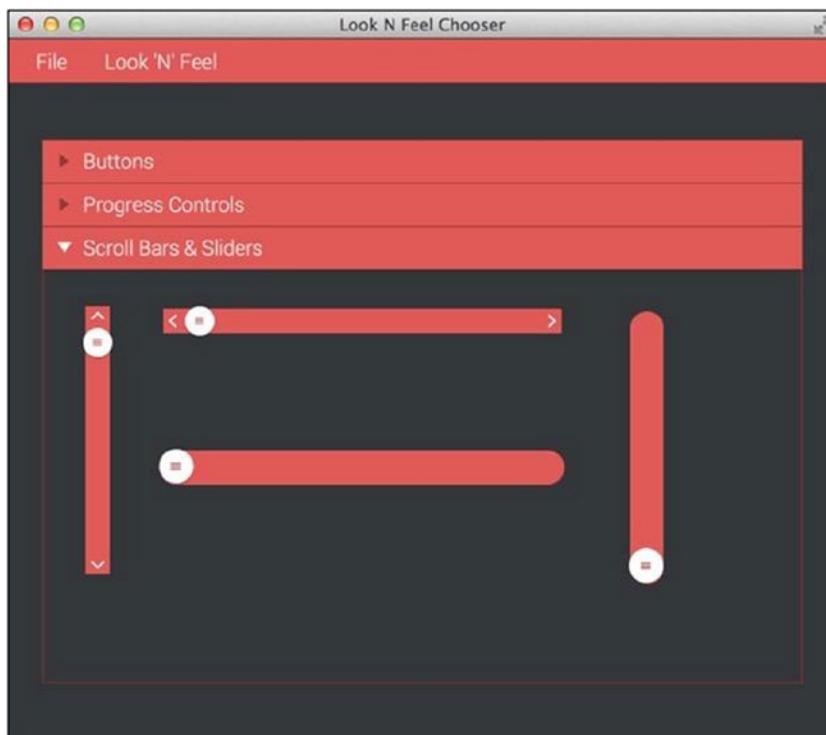


Figure 15-9. The Look N Feel Chooser application using the Flat Red look and feel

In Figure 15-9, notice that the accordion UI control's title bar says Scroll Bars & Sliders. It's expanded with UI controls sporting the selected look and feel. This is a great way to show off different look and feel themes to your customers or users while their application continues to function as usual.

The next section explains the main source code of the Look N Feel Chooser application in Listing 15-2. For the sake of space, you see the `start()` method and not the entire source code. To see the entire source code listing of the Look N Feel Chooser application, visit the book's web site to download the project under Chapter 15's `LookAndFeelChooser`.

The Look N Feel Chooser Example Application Code

By loading the Look N Feel Chooser project into the NetBeans IDE, you can see that the source code consists of six files: `LookAndFeelChooser.java`, `lnf_demo.fxml`, `controlStyle1.css`, `controlStyle2.css`, `flatred.css`, and `sky.css`. The `.ttf` files are TrueType fonts that are used in the Flat Red look and feel. Figure 15-10 shows the project structure in the NetBeans IDE.

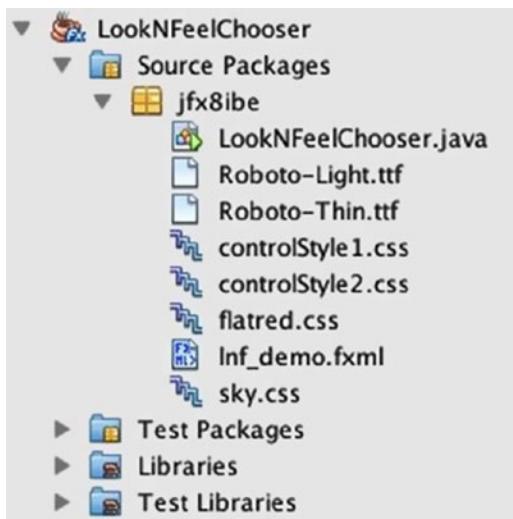


Figure 15-10. The Look N Feel Chooser project structure in the NetBeans IDE

Looking at the project structure, you'll notice the `LookNFeelChooser.java` file as the main driver application class. The `lnf_demo.fxml` file is an FXML-formatted file representing the center content, which contains the accordion UI control with other UI control elements. The FXML file was created by the Scene Builder tool. The rest of the files are JavaFX CSS style sheets that represent different look-and-feel themes: `controlStyle1.css`, `controlStyle2.css`, `flatred.css`, and `sky.css`.

Listing 15-2 shows the `init()` and `start()` methods of the `LookNFeelChooser.java` Application class. Remember that all JavaFX applications are first initialized via the `init()` method and then begin with the `start()` method. After you have examined Listing 15-2, you can read about how it all works.

Listing 15-2. The Look N Feel Chooser Project Source Code to Switch Between Look and Feel (JavaFX CSS Style Sheets. (`LookNFeelChooser.java`)

```

@Override public void init() {
    Font.loadFont(LookNFeelChooser.class
                  .getResourceAsStream("Roboto-Thin.ttf"), 10)
                  .getName();
    Font.loadFont(LookNFeelChooser.class
                  .getResourceAsStream("Roboto-Light.ttf"), 10)
                  .getName();
}

@Override public void start(Stage primaryStage) throws IOException {
    BorderPane root = new BorderPane();
    Parent content = FXMLLoader.load(getClass().getResource("lnf_demo.fxml"));
    Scene scene = new Scene(root, 650, 550, Color.WHITE);
    root.setCenter(content);

    // Menu bar
   MenuBar menuBar = newMenuBar();

    // File menu
    Menu fileMenu = new Menu("_File");

```

```
MenuItem exitItem = new MenuItem("Exit");
exitItem.setAccelerator(new KeyCodeCombination(KeyCode.X, KeyCombination.SHORTCUT_DOWN));
exitItem.setOnAction(ae -> Platform.exit());

fileMenu.getItems().add(exitItem);
menuBar.getMenus().add(fileMenu);

// Look and feel menu
Menu lookNFeelMenu = new Menu("_Look 'N' Feel");
lookNFeelMenu.setMnemonicParsing(true);
menuBar.getMenus().add(lookNFeelMenu);
root.setTop(menuBar);

// Look and feel selection
MenuItem caspianMenuItem = new MenuItem("Caspian");
caspianMenuItem.setOnAction(ae -> {
    scene.getStylesheets().clear();
    setUserAgentStylesheet(null);
    setUserAgentStylesheet(STYLESHEET_CASPIAN);
});

MenuItem modenaMenuItem = new MenuItem("Modena");
modenaMenuItem.setOnAction(ae -> {
    scene.getStylesheets().clear();
    setUserAgentStylesheet(null);
    setUserAgentStylesheet(STYLESHEET_MODENA);
});

MenuItem style1MenuItem = new MenuItem("Control Style 1");
style1MenuItem.setOnAction(ae -> {
    scene.getStylesheets().clear();
    setUserAgentStylesheet(null);
    scene.getStylesheets()
        .add(getClass().getResource("controlStyle1.css")
            .toExternalForm());
});

MenuItem style2MenuItem = new MenuItem("Control Style 2");
style2MenuItem.setOnAction(ae -> {
    scene.getStylesheets().clear();
    setUserAgentStylesheet(null);
    scene.getStylesheets()
        .add(getClass().getResource("controlStyle2.css")
            .toExternalForm());
});

MenuItem skyMenuItem = new MenuItem("Sky LnF");
skyMenuItem.setOnAction(ae -> {
    scene.getStylesheets().clear();
    setUserAgentStylesheet(null);
    scene.getStylesheets()
```

```

        .add(getClass().getResource("sky.css")
            .toExternalForm());
    });

MenuItem flatRedMenuItem = new MenuItem("FlatRed");
flatRedMenuItem.setOnAction(ae -> {
    scene.getStylesheets().clear();
    setUserAgentStylesheet(null);
    scene.getStylesheets()
        .add(getClass().getResource("flatred.css")
            .toExternalForm());
});

lookNFeelMenu.getItems()
    .addAll(caspianMenuItem,
        modenaMenuItem,
        style1MenuItem,
        style2MenuItem,
        skyMenuItem,
        flatRedMenuItem);

primaryStage.setTitle("Look N Feel Chooser");
primaryStage.setScene(scene);
primaryStage.show();
}

```

Note To run this example, make sure the CSS and TTF (true type font) files are in the compiled class's area. Resource files can be loaded easily when placed in the same directory (package) as the compiled class file that is loading them. The CSS files are initially co-located with this source code example file. In NetBeans, you can select Clean and Build Project or you can copy your files to your class's build area.

How It Works

The Look N Feel Chooser application is first initialized with the overloaded method `init()`, which loads the required fonts used for the Flat Red look-and-feel theme. After the `init()` method is executed, the JavaFX application lifecycle will invoke the `start()` method.

In the `start()` method, the code begins by creating a border pane layout, which then loads an FXML file to be placed as the center content region. FXML is an XML-based language to express JavaFX UIs. This provides a way to separate the presentation layer from the application logic layer. Typically, FXML is generated by a GUI builder tool, which allows a designer to drag and drop controls and create UIs graphically. The following line uses the `FXMLLoader.load()` method to unmarshal (deserialize) the center content. The center content pane is an `AnchorPane` layout containing an `Accordion` control holding other UI controls.

```
Parent content = FXMLLoader.load(getClass().getResource("lnf_demo.fxml"));
```

To learn more about the Scene Builder tool, see Chapter 5.

Continuing with the example code, you will notice the usual creation of a scene with a root node. Next, the code builds a menu bar with menu items. The first menu is the File menu, which allows users to exit the application. Notice that for a quick exit using a keyboard shortcut, the KeyCodeCombination instance allows the users to press the Ctrl+X key combo. To learn about menus and keyboard shortcuts, refer to Chapter 4.

The second menu option on the Look N Feel allows the users to select predefined look and feel CSS style sheet files. The menu items are set to invoke handler code (lambda expressions) that is triggered based on an onAction event. The following code snippet is a menu item that is responsible for switching the application's look and feel with the Caspian CSS style:

```
MenuItem caspianMenuItem = new MenuItem("Caspian");
caspianMenuItem.setOnAction(ae -> {
    scene.getStylesheets().clear();
    setUserAgentStylesheet(STYLESHEET_CASPIAN);
});
```

Above the JavaFX API, the STYLESHEET_CASPIAN and STYLESHEET_MODENA values are strings that represent the CSS files based on their location on the classpath.

Because the rest of the code is pretty similar, I'm going to fast-forward to the last look and feel menu option, which switches to the Flat Red look and feel. The other menu items are essentially identical and invoke the same methods to clear previously loaded CSS style sheet files. However, for the Flat Red look and feel, I used the getStylesheets().add() method to style only the current scene-level nodes.

The Flat Red look and feel styles a subset of UI controls that includes sliders, scroll bars, buttons, and progress controls. The code snippet shown next sets up a menu item to be selected that will skin the application with the Flat Red look and feel. The onAction code clears the scene's style sheets and then sets the UserAgentStylesheet to null. Lastly, the flatred.css file is loaded to style the given scene.

```
MenuItem flatRedMenuItem = new MenuItem("FlatRed");
flatRedMenuItem.setOnAction(ae -> {
    scene.getStylesheets().clear();
    setUserAgentStylesheet(null);
    scene.getStylesheets()
        .add(getClass().getResource("flatred.css")
            .toExternalForm());
});
```

JavaFX CSS Styling

Now that you know how to load CSS style sheet files, let's discuss JavaFX CSS selectors and styling properties (rules). Similar to the way HTML5 uses CSS style sheets, there are selectors or style classes associated with Node objects on the Scene graph. All JavaFX Scene graph nodes have a setId(), a getClass().getStyleClass().add(), and a setStyle() method to apply styling properties that could potentially change the node's background color, border, stroke, and so on.

Before you learn about selectors, I want to refer you to the *JavaFX CSS Reference Guide* at the following location:

<http://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html>

This invaluable reference guide will be very handy throughout this book and beyond.

What Are Selectors?

Similar to the W3C CSS (World Wide Web Consortium) standards for styling HTML elements, JavaFX CSS has the concept of selectors. Selectors are basically tags to help locate JavaFX nodes on the Scene graph to be styled using CSS style definitions. The two kinds of selector types are `id` and `class`. An `id` selector is a unique string name that is set on a scene node.

A `class` selector is also a string name that can be added as a tag to any JavaFX node. I should also point out that a `class` selector has no relation to the concepts of C++ or Java classes. Class selectors allow any number of nodes to contain the same class string name for the ability to style nodes with one CSS style definition. In the next sections, you learn more about how to define CSS selector types, and I follow up with an example.

CSS id Type Selectors

The `id` type selector is a *unique* string name assigned to a node. This means that no other node's ID can be the same. When using `id` type selectors, you invoke the `setId(String ID)` method on a JavaFX node object. For example, to target a `Button` instance whose ID is `my-button`, you invoke the `setId("my-button")` method. To style the button with an ID of `my-button`, you create a CSS style definition block declared with an `ID` selector `#my-button`, as shown:

```
#my-button {
    -fx-text-fill: rgba(17, 145, 213, .90);
    -fx-border-color: rgba(255, 255, 255, .80);
    -fx-border-radius: 8;
    -fx-padding: 6 6 6 6;
    -fx-font: bold italic 20pt "LucidaBrightDemiBold";
}
```

This CSS styling block will be applied to a button with the *unique ID* of `my-button`. Thus no other node will be allowed to contain an ID of `my-button`. You will also notice that when you use `ID` selectors in styling blocks, the CSS selector name is prefixed with the `#` symbol, and when setting the `id` in Java code, the `#` symbol is not used.

CSS class Type Selectors

When using `class` type selectors, you will be invoking the `getStyleClass().add(String styleClass)` method to add a *selector* to a node. The method allows you to have multiple style classes for styling a node. Since the `getStyleClass()` method returns an `ObservableList`, you can add and remove style classes in an ad hoc fashion to update their appearance dynamically. For example, let's target two buttons whose style classes (`ObservableList`) contain a class called `num-button` via the `getStyleClass().add("num-button")` method. The following is a CSS style definition block declared with a `class` selector `.num-button`:

```
.num-button {
    -fx-background-color: white, rgb(189,218,230), white;
    -fx-background-radius: 50%;
    -fx-background-insets: 0, 1, 2;
    -fx-font-family: "Helvetica";
    -fx-text-fill: black;
    -fx-font-size: 20px;
}
```

This CSS styling block will be applied to buttons with a style class `num-button`. You will notice that when using class selectors, the CSS selector name is prefixed with a dot (.), and when adding the selector in Java code, the (.) symbol is not present.

Selector Patterns

Up until now, you've seen only simple selectors; however, selectors can have patterns that traverse the Scene graph's hierarchy of nodes from the root node to the child nodes. A full discussion of selector patterns is beyond the scope of this book, but I briefly mention common selector patterns and introduce pseudo-classes.

Common Selector Patterns

Often you will want to style many nodes based on common selector patterns. A typical selector pattern you might want to perform is to style child nodes whose parent is of a certain type. For example, you might want to style all buttons whose parent is an `HBox`.

Another pattern would be to style two different kinds of nodes with a common property. The following example shows selector patterns for both of these use cases:

```
/* style all buttons who's parent is an HBox */
.hbox > .button {
    -fx-text-fill: black;
}

/* style all labels and text nodes */
.label, .text {
    -fx-font-size: 20px;
}
```

As you can see, the selector pattern (`.hbox > .button`) styles `Button` nodes that are descendants of an `HBox`. This is pretty straightforward. The *greater-than* symbol between the two selectors lets the system know which nodes to style.

Also, as described in the *JavaFX CSS Reference Guide*, the selectors for UI controls have a naming convention. They are all lowercase letters, and if a control has more than one word, they are separated by hyphens. For instance, a `GridPane`'s class selector would be `.grid-pane`. Keep in mind that *not* all nodes have named class selectors by default, so refer to the *JavaFX CSS Reference Guide*.

The second use case with the selector pattern (`.label, .text`) has to do with setting a *common* property that is shared by different types of nodes. This example is a selector pattern that styles all `Label` nodes and `Text` nodes to have the same text font size. The *comma* denotes a list of selectors to style. In other words, in this scenario the font size will be styled to be 20 points.

Pseudo-Class Selectors

Pseudo-class selectors are used to style nodes that have different states. An example is a `Button` node's hover state. A button control has the following states: `armed`, `disabled`, `focused`, `hover`, `pressed`, and `show-mnemonic`. To specify selectors with pseudo-classes you must append a colon and the state (type) to the main selector name. The following code snippet shows two selector patterns for a button that has a class selector `num-button`:

```
.num-button {
    -fx-background-color: white, bluish-gray, white;
    -fx-background-radius: 50%;
    -fx-background-insets: 0, 1, 2;
```

```
-fx-font-family: "Helvetica";
-fx-text-fill: black;
}
.num-button:hover {
    -fx-background-color: black, white, black;
    -fx-text-fill: white;
}
```

In this CSS code, the first (`.num-button`) selector merely styles a button in its normal state. However, the second selector with the appended colon and state (`:hover`) will change a subset of properties, altering them slightly. In this scenario, when a mouse cursor moves over (hovers) a button with the style class `num-button:hover`, the color of the button and the text's fill color will be reversed. Figure 15-11 shows the pseudo-selector in action when the mouse is hovering over the number pad on button 3.

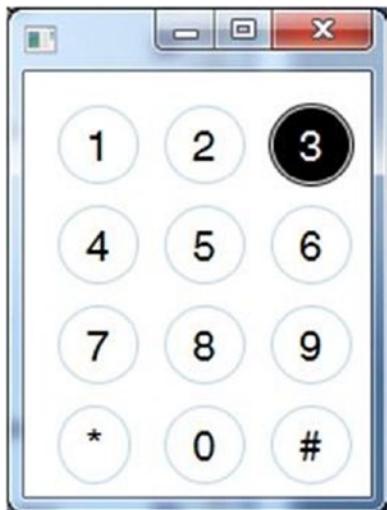


Figure 15-11. A selector with a pseudo-class of `hover` is used when the mouse cursor is on top of any number pad button

A Selector Styling Example

To demonstrate selectors using `id` and `class` type selectors, I created an application resembling a smartphone's number pad, as shown in Figure 15-12.



Figure 15-12. An application to mimic a number pad from a smartphone.

In Figure 15-12, notice the 4×3 grid of round buttons for a number pad and a green rectangular button beneath it to make a phone call. Listing 15-3 is the main `start()` method containing the source code. Listing 15-4 then shows the contents of the CSS file `mobile_buttons.css`.

Listing 15-3. The JavaFX Source Code for the Number Pad Application

```

@Override
public void start(Stage primaryStage) {
    BorderPane root = new BorderPane();
    Scene scene = new Scene(root, 180, 250);
    scene.getStylesheets()
        .add(getClass().getResource("mobile_buttons.css")
            .toExternalForm());
    String[] keys = {"1", "2", "3",
                    "4", "5", "6",
                    "7", "8", "9",
                    "*", "0", "#"};
    GridPane numPad = new GridPane();
    numPad.getStyleClass().add("num-pad");
    for (int i=0; i < 12; i++) {
        Button button = new Button(keys[i]);
        button.getStyleClass().add("num-button");
        numPad.add(button, i % 3, (int) Math.ceil(i/3) );
    }
    // Call button
    Button call = new Button("Call");
    call.setId("call-button");
    call.setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
    numPad.add(call, 0, 4);
    GridPane.setColumnSpan(call, 3);
    GridPane.setHgrow(call, Priority.ALWAYS);
    root.setCenter(numPad);
    primaryStage.setScene(scene);
    primaryStage.show();
}

```

The code in Listing 15-3 will load the `mobile_button.css` file that gets applied to the Scene graph. The CSS file's contents are shown in Listing 15-4.

Listing 15-4. The Contents of the `mobile_buttons.css` CSS File

```
.root {
    -fx-background-color: white;
    -fx-font-size: 20px;
    bright-green: rgb(59,223, 86);
    bluish-gray: rgb(189,218,230);
}

.num-pad {
    -fx-padding: 15px, 15px, 15px, 15px;
    -fx-hgap: 10px;
    -fx-vgap: 8px;
}

.num-button {
    -fx-background-color: white, bluish-gray, white;
    -fx-background-radius: 50%;
    -fx-background-insets: 0, 1, 2;
    -fx-font-family: "Helvetica";
    -fx-text-fill: black;
}

.num-button:hover {
    -fx-background-color: black, white, black;
    -fx-text-fill: white;
}

#call-button {
    -fx-background-color: white, bright-green;
    -fx-background-radius: 2;
    -fx-background-insets: 0, 1;
    -fx-font-family: "Helvetica";
    -fx-text-fill: white;
}

#call-button:hover {
    -fx-background-color: bright-green, white;
    -fx-background-radius: 2;
    -fx-background-insets: 0, 1;
    -fx-font-family: "Helvetica";
    -fx-text-fill: bright-green;
}
```

How It Works

The code starts by creating a scene that has a `BorderPane` as a root node. After the scene is created, the code loads the CSS style sheet file `mobile_buttons.css` to style the current scene's nodes. Next, the code simply creates a grid using the `GridPane` class and generates 12 buttons to be placed in each cell. Notice in the `for` loop that each button is set with the style class named `num-button` via the `getStyleClass().add()` method.

Last, the green call button will be added to the last row of the grid pane. Because the call button is unique, its `id` selector is set with `call-button`, and it is styled using the `id` selector, which means the selector

named inside the CSS file will be prefixed with the # symbol. To see some amazing button styling, head over to the FXExperience.com's blog entry on styling buttons at the following link:

<http://fxexperience.com/2011/12/styling-fx-buttons-with-css>

At FXExperience, the blog entry on styling FX buttons is by Jasper Potts (developer experience architect on client java at Oracle).

How to Define -fx- Based Styling Properties (Rules)

I'm sure by now you have noticed the many name-value pairs (*properties*) inside styling definition blocks having the prefix -fx-. These properties, often called rules, can be defined to have values that can set a region's border width, background fill colors, and so on. In this section, you learn how to style a JavaFX node using selector styling blocks and the ability to override properties by using inline styling.

Styling a Node with a Selector Style Definition Block

A selector style definition block always begins with a selector name prefixed with a . or # symbol. The symbol determines the selector type as mentioned earlier. A selector will start a block with an open curly brace, followed by property or rule definitions. Each JavaFX theme property or rule will be prefixed with -fx- and its appropriate property name. A property name and value are separated by a : symbol, and the pair ends with a semicolon. To finish the styling block, you simply end the block with a closing curly brace. The syntax for a selector style definition block looks like this:

```
. or #<selector-name> <pattern>{
    -fx-<some-property> : <some-value>;
}
```

One last thing to mention is the ability to add comments to CSS styling definitions. To add comments, you use an opening slash asterisk /* and a closing asterisk slash */ (the same as adding comments in C/C++ and Java). Shown next is an example of using comments in a selector style block definition:

```
.num-button {
    -fx-background-color: white, rgb(189,218,230), white;
    /* This is a comment.
       -fx-background-radius: 50%;
    */
    -fx-background-insets: 0, 1, 2;
    -fx-font-family: "Helvetica";
    -fx-text-fill: black;
    -fx-font-size: 20px;
}
```

Styling a Node by Inlining JavaFX CSS Styling Properties

While CSS selector blocks are the recommended way to style your JavaFX nodes, there might be situations where you will want to override styling properties. For example, you may want to enlarge a button and change its text color temporarily as a mouse cursor hovers (OnMouseEntered) over the specified button. When the mouse cursor is not hovering (OnMouseExited) the button's inline style is removed, which then reverts the styling to the parent class or id selector styling blocks. To override a node's styling property set from its ancestor (parent) id or class selector, you can invoke the node's method `setStyle()`.

To implement the prior example just mentioned, the following code snippet implements a button with handler code that responds to `OnMouseEntered` and `OnMouseExited` events and toggles the button's text size and color.

```
Button button = new Button("Press Me");
button.getStyleClass()
    .add("my-default-style");

button.setOnMouseEntered(actionEvent ->
    button.setStyle("-fx-font-size: 30px; -fx-text-fill: green;"));

button.setOnMouseExited(actionEvent -> button.setStyle(""));
```

Styling Properties (Rules) Limitations

Because all graph nodes extend from the `Node` class, derived classes will be able to inherit styling properties from their ancestors. Knowing the inheritance hierarchy of node types is very important because the type of node will help determine the types of styling properties you can control. For instance, a `Rectangle` class extends from `Shape`, which extends from `Node`. Having said this, you will later see that some properties that you assumed would be on a node don't exist. So how do you know what properties exist? The solution is to refer to the *JavaFX CSS Reference Guide*. Again, based on the type of node, there are limitations to the styles you are able to set.

Obeying the JavaFX CSS Rules

Did you know that styling properties (rules) can be overridden? To override properties, you must learn about the order of precedence when rules are defined. Figure 15-13 depicts a typical Scene graph with a root (parent) node and child nodes. A root node such as a `BorderPane` layout node will contain a *class* selector of `root`, which is meant for top-level properties.

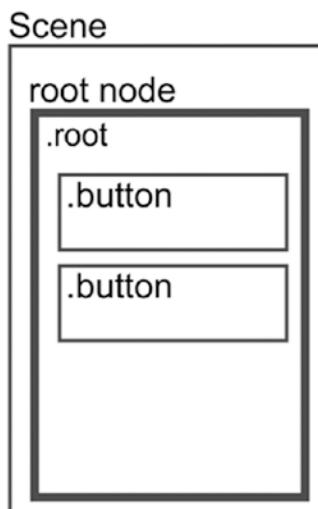


Figure 15-13. A scene with a root node that has buttons as child nodes

These are basically properties common to many nodes, such as font size for text or background color. This is a great feature for theming, because many nodes that share a property will inherit that change. As mentioned earlier, some nodes will contain default selectors; for example, JavaFX Button instances have a `.button` class selector.

In Figure 15-13, the class selector `root` could have a property that is common across many nodes related to text:

```
.root {
    -fx-font-size: 12px;
}
.button {
    -fx-font-size: 20px;
}
```

When you want to style a button's text font size, you can override the root's styling definition. In order to override the `-fx-font-size` property, the button style class definition block will override the 12 point font from the parent with a 20 point font.

Believe it or not, there is another way to override the style definition block of the class selector `.button` in addition to the method just shown. Similar to HTML5 CSS, each element has a *style attribute*. JavaFX's graph nodes also have a style attribute (property). Following is an example of how to set the style property programmatically in Java code:

```
Button button = new Button("press it!");
button.setStyle("-fx-font-size: 30px;");
```

This code will override any `root-`, `id-`, or `class-level` selector style definition block. Basically, the invocation to the `setStyle()` method allows you to add any number of property-value pairs as long they are separated by semicolons.

Custom Controls

“Use the force, Luke.”

—Obi-Wan Kenobi

With all of your newfound abilities, there is yet another powerful API you will want to harness. Imagine a customer whose requirements call for specialized UI controls such as LEDs, gauges, knobs, and light indicators. Surely you could attempt to use the existing UI controls to be styled, but it is rather difficult, if not impossible. So, what is a young Padawan to do? (Now, say the quote). Welcome to the world of JavaFX custom controls (the *Controls API*)!

As an example of creating a custom control, in this section you learn how to create a light-emitting diode (LED) control. The section begins with a description of the LED control. Then you see how the code is structured as you look at the classes involved in the creation of custom controls. After that, you get a chance to look at the code before you walk through the code details.

The LED Custom Control

The light-emitting diode (LED) is something that you will find in many electronic devices to indicate status (such as on or off). So why not use an LED as a custom control in your code as a visual indicator? Typically, an LED contains two wires and a body made from clear plastic, as shown in Figure 15-14.



Figure 15-14. A typical light-emitting diode (LED)

Often the LED is mounted to a socket. To create a custom JavaFX control to emulate a physical object, you first have to decide which materials and components you will portray.

In the case of the LED custom JavaFX control, we can approximate this appearance:

- Metal socket
- Plastic body
- Light effect on top of the plastic body due to the curved surface

One thing that Java developers often don't like, but which is really helpful for designing a custom control, is using a drawing program. To be precise, a vector drawing program. When the appearance of your control is important, it makes sense to use a tool that is good for visualizing things—a drawing program, such as the one used to create Figure 15-15. So the first thing you should do is create a vector drawing of the control you plan to create.



Figure 15-15. A vector drawing of the LED control

You can see the metal socket around the red plastic body and the white highlight. The red body contains an inner- and outer-shadow to create a more realistic look. If you look at the parts of the vector drawing, you will find three circles filled with gradients, as shown in Figure 15-16.

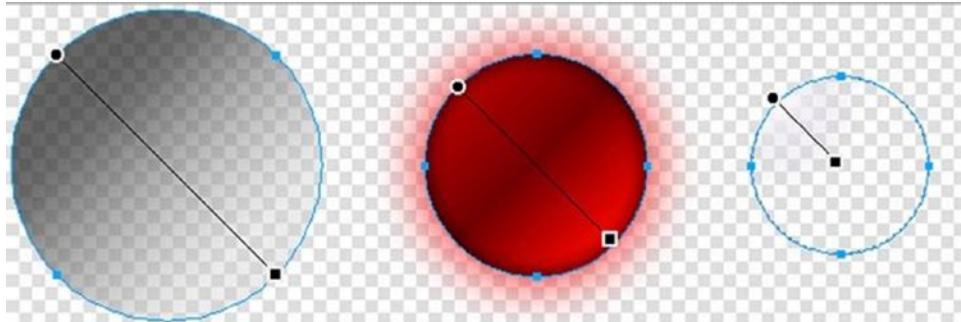


Figure 15-16. The three parts of the vector LED

You start coding in the drawing program because here you define the size, colors, gradients, and positions of your control. The big advantage of using a drawing program is the direct visual feedback that you get by changing parameters like color, size, and position.

In JavaFX there's not only one way of creating a custom control, but many. Here is a list of valid approaches:

- Customize the CSS of an existing control
- Extend an existing control
- Extend the Region node
- Create a control, a skin, and a CSS file
- Use the Canvas node

This chapter focuses on the approach that extends a Region node and uses CSS. To get an idea of how to create the same LED control by using the Canvas node or by using a separate Control class, Skin class, and a CSS file, look at the GitHub repository at <http://github.com/HanSolo/JFX8CustomControls>. For this book, you can get the source code from the book's web site or from the following:

<https://github.com/carldea/jfx9be/blob/master/chap15/JFX9CustomControls/src/jfx9controls/ledregion>

Structure of the LED Custom Control Example Code

Usually I structure the code in my controls in the following way:

- Constructor
- Initialization:
 - An `init()` method defines the initial size.
 - An `initGraphics()` method sets up the Scene graph of the control.
 - A `registerListeners()` method hooks up listeners to properties.

- A Methods block contains the get, set, and property methods.
- A Resizing block contains methods to resize and redraw the control (if needed).

The Properties of the LED Control

The LED control will contain the logic of the control (its properties) and the visualization code. Because the LED is a very simple control, you do not need many properties. This example uses five properties:

- On (Boolean property for current state)
- Blinking (Boolean property to switch on/off blinking)
- Interval (long property to define the blink interval)
- FrameVisible (Boolean property to switch on/off the metal socket)
- LedColor (object property of type Color to define the color of the LED)

[Listing 15-5](#) shows the get, set, and property methods for these properties.

Listing 15-5. Property Manipulation Code (Led.java)

```
public class Led extends Region {
    private static final double PREFERRED_SIZE = 16;
    private static final double MINIMUM_SIZE = 8;
    private static final double MAXIMUM_SIZE = 1024;
    private static final PseudoClass ON_PSEUDO_CLASS = PseudoClass.getPseudoClass("on");
    private static final long SHORTEST_INTERVAL = 50_000_000L;
    private static final long LONGEST_INTERVAL = 5_000_000_000L;

    // Model/Controller related
    private ObjectProperty<Color> ledColor;
    private BooleanProperty on;
    private boolean _blinking = false;
    private BooleanProperty blinking;
    private boolean _frameVisible = true;
    private BooleanProperty frameVisible;
    private long lastTimerCall;
    private long _interval = 500_000_000L;
    private LongProperty interval;
    private AnimationTimer timer;

    // View related
    private double size;
    private Region frame;
    private Region led;
    private Region highlight;
    private InnerShadow innerShadow;
    private DropShadow glow;

    public final boolean isOn() {
        return null == on ? false : on.get();
    }
    public final void setOn(final boolean ON) {
        onProperty().set(ON);
    }
}
```

```
public final BooleanProperty onProperty() {
    if (null == on) {
        on = new BooleanPropertyBase(false) {
            @Override protected void invalidated() { pseudoClassStateChanged(ON_PSEUDO_
                CLASS, get()); }
            @Override public Object getBean() { return this; }
            @Override public String getName() { return "on"; }
        };
    }
    return on;
}

public final boolean isBlinking() {
    return null == blinking ? _blinking : blinking.get();
}
public final void setBlinking(final boolean BLINKING) {
    if (null == blinking) {
        _blinking = BLINKING;
        if (BLINKING) {
            timer.start();
        } else {
            timer.stop();
            setOn(false);
        }
    } else {
        blinking.set(BLINKING);
    }
}
public final BooleanProperty blinkingProperty() {
    if (null == blinking) {
        blinking = new BooleanPropertyBase() {
            @Override public void set(final boolean BLINKING) {
                super.set(BLINKING);
            }
            if (BLINKING) {
                timer.start();
            } else {
                timer.stop();
                setOn(false);
            }
        };
        @Override public Object getBean() {
            return Led.this;
        }
        @Override public String getName() {
            return "blinking";
        }
    };
}
return blinking;
}
```

```

public final long getInterval() {
    return null == interval ? _interval : interval.get();
}
public final void setInterval(final long INTERVAL) {
    if (null == interval) {
        _interval = clamp(SHORTEST_INTERVAL, LONGEST_INTERVAL, INTERVAL);
    } else {
        interval.set(INTERVAL);
    }
}
public final LongProperty intervalProperty() {
    if (null == interval) {
        interval = new LongPropertyBase() {
            @Override public void set(final long INTERVAL) {
                super.set(clamp(SHORTEST_INTERVAL, LONGEST_INTERVAL, INTERVAL));
            }
            @Override public Object getBean() {
                return Led.this;
            }
            @Override public String getName() {
                return "interval";
            }
        };
    }
    return interval;
}

public final boolean isFrameVisible() {
    return null == frameVisible ? _frameVisible : frameVisible.get();
}
public final void setFrameVisible(final boolean FRAME_VISIBLE) {
    if (null == frameVisible) {
        _frameVisible = FRAME_VISIBLE;
    } else {
        frameVisible.set(FRAME_VISIBLE);
    }
}
public final BooleanProperty frameVisibleProperty() {
    if (null == frameVisible) {
        frameVisible = new SimpleBooleanProperty(this, "frameVisible", _frameVisible);
    }
    return frameVisible;
}

public final Color getLedColor() {
    return null == ledColor ? Color.RED : ledColor.get();
}
public final void setLedColor(final Color LED_COLOR) {
    ledColorProperty().set(LED_COLOR);
}
public final ObjectProperty<Color> ledColorProperty() {
    if (null == ledColor) {

```

```

        ledColor = new SimpleObjectProperty<>(this, "ledColor", Color.RED);
    }
    return ledColor;
}
// **** Utility Methods ****
public static long clamp(final long MIN, final long MAX, final long VALUE) {
    if (VALUE < MIN) return MIN;
    if (VALUE > MAX) return MAX;
    return VALUE;
}

```

The Initialization Code of the LED Control

The first thing you have to do in the *constructor* of the LED control is load the corresponding CSS file and add the main style class as follows:

```

public Led() {
    getStylesheets().add(getClass().getResource("led.css").toExternalForm());
    getStyleClass().add("led");
}
Then we simply initialize the AnimationTimer that we will use to make the LED control blink:
lastTimerCall = System.nanoTime();
timer = new AnimationTimer() {
    @Override public void handle(final long NOW) {
        if (NOW > lastTimerCall + getInterval()) {
            setOn(!isOn());
            lastTimerCall = NOW;
        }
    }
}

```

The last thing you have to do in the constructor is call these methods to initialize the size of the control, initialize the Scene graph, and register some listeners:

```

init();
initGraphics();
registerListeners();

```

To make sure that the LED control will be correctly sized during initialization, you must set the minimum, preferred, and maximum sizes in the *init()* method, which looks as follows:

```

private void init() {
    if (Double.compare(getWidth(), 0) <= 0 || Double.compare(getHeight(), 0) <= 0 ||
        Double.compare(getPrefWidth(), 0) <= 0 || Double.compare(getPrefHeight(), 0) <= 0) {
        setPreferredSize(PREFERRED_SIZE, PREFERRED_SIZE);
    }
    if (Double.compare(getMinWidth(), 0) <= 0 || Double.compare(getMinHeight(), 0) <= 0) {
        setMinSize(MINIMUM_SIZE, MINIMUM_SIZE);
    }
    if (Double.compare(getMaxWidth(), 0) <= 0 || Double.compare(getMaxHeight(), 0) <= 0) {
        setMaxSize(MAXIMUM_SIZE, MAXIMUM_SIZE);
    }
}

```

Visualization Code

The `javafx.scene.layout.Region` node is a lightweight JavaFX container that can contain other nodes and be styled by CSS. By extending the `Region` node, the custom control will contain the logic of the control and also the visualization code. In the `initGraphics()` method shown in Listing 15-6, the code sets up the Scene graph for the control by creating the nodes that are needed and applying the appropriate CSS styles to them.

Listing 15-6. Setting Up the LED Control Scene Graph

```
private void initGraphics() {
    // Create the node for the metal socket
    frame = new Region();
    frame.getStyleClass().setAll("frame");
    frame.setOpacity(isFrameVisible() ? 1 : 0);
    // Create the node for the main LED plastic body
    led = new Region();
    led.getStyleClass().setAll("main");
    led.setStyle("-led-color: " + (getLedColor()).toString().replace("0x", "#") + ";");
    // Create the inner shadow effect for the main LED body
    innerShadow = new InnerShadow(BlurType.TWO_PASS_BOX,
        Color.rgb(0, 0, 0, 0.65),
        8, 0d, 0d, 0d);
    // Create the drop shadow effect for the main LED body (the glow effect)
    glow = new DropShadow(BlurType.TWO_PASS_BOX,
        getLedColor(),
        20, 0d, 0d, 0d);
    glow.setInput(innerShadow);
    // Create the node for the highlight effect on the main LED body
    highlight = new Region();
    highlight.getStyleClass().setAll("highlight");
    // Add all nodes to the Scene graph of this control
    getChildren().addAll(frame, led, highlight);
}
```

So far, the example has created each node that you need for the LED control and applied the appropriate style from the CSS file.

The LED Control CSS File

Each node that is created in the `initGraphics()` method gets its own CSS style class, which can be found in the `led.css` file. It looks like this:

```
/* The main led style class where the -led-color variable is defined */
.led {
    -led-color : red;
    -frame-color: linear-gradient(from 14% 14% to 84% 84%,
        rgba(20, 20, 20, 0.64706) 0%,
        rgba(20, 20, 20, 0.64706) 15%,
        rgba(41, 41, 41, 0.64706) 26%,
        rgba(200, 200, 200, 0.40631) 85%,
        rgba(200, 200, 200, 0.3451) 100%);
```

```

/* The .frame sub-class, which defines the fill for the metal socket */
.led .frame {
    -fx-background-color : -frame-color;
    -fx-background-radius: 1024;
}
/* The .main sub-class, which defines the fill for the LED plastic body when it's off */
.led .main {
    -fx-background-color : linear-gradient(from 15% 15% to 83% 83%,
                                              derive(-led-color, -80%) 0%,
                                              derive(-led-color, -87%) 49%,
                                              derive(-led-color, -80) 100%);
    -fx-background-radius: 1024;
}
/* The .main sub class with pseudo-class :on that defines the fill for the LED plastic
   body when it's on
*/
.led:on .main {
    -fx-background-color: linear-gradient(from 15% 15% to 83% 83%,
                                              derive(-led-color, -23%) 0%,
                                              derive(-led-color, -50%) 49%,
                                              -led-color 100%);
}
/* The .highlight sub-class that defines the fill of the highlight effect */
.led .highlight {
    -fx-background-color : radial-gradient(center 15% 15%, radius 50%,
                                              white 0%,
    -fx-background-radius: 1024;
}

```

Because in CSS, you can use percentage to define positions in gradients, you don't have worry about the real size of the control to calculate the start and stop positions of the gradients. That is a huge advantage compared to Java Swing, where you had to calculate all these values every time the size of the control changed. So all these calculations will be done automatically by JavaFX. That reduces the resizing code a lot, as you will see.

Resizing the LED Control

In JavaFX, the size of the layout container determines the size of its children, so you have to ensure that your control is resized by its layout container. That means if you put your LED control in a StackPane (which resizes its children relative to its own size), the LED will be sized the same as the StackPane. Therefore, you must hook up listeners to the `widthProperty()` and `heightProperty()` of the control in the `registerListeners()` method.

```

private void registerListeners() {
    widthProperty().addListener(observable -> resize());
    heightProperty().addListener(observable -> resize());
    frameVisibleProperty().addListener(observable ->
        frame.setOpacity(isFrameVisible() ? 1 : 0));
    onProperty().addListener(observable -> led.setEffect(isOn() ? glow : innerShadow));
    ledColorProperty().addListener(observable -> {
        led.setStyle("-led-color: " + (getLedColor()).toString().replace("0x", "#") + ";");
        resize();
    });
}

```

This method hooks up listeners to all properties that might have an effect on either visualization or size of the LED control. When the listeners for the width and height are triggered, they will call the `resize()` method, and that will take care of sizing all nodes in the control.

```
private void resize() {
    size = getWidth() < getHeight() ? getWidth() : getHeight();
    if (size > 0) {
        if (getWidth() > getHeight()) {
            setTranslateX(0.5 * (getWidth() - size));
        } else if (getHeight() > getWidth()) {
            setTranslateY(0.5 * (getHeight() - size));
        }
        innerShadow.setRadius(0.07 * size);
        glow.setRadius(0.36 * size);
        glow.setColor(getLedColor());
        frame.setPrefSize(size, size);
        led.setPrefSize(0.72 * size, 0.72 * size);
        led.relocate(0.14 * size, 0.14 * size);
        led.setEffect(isOn() ? glow : innerShadow);
        highlight.setPrefSize(0.58 * size, 0.58 * size);
        highlight.relocate(0.21 * size, 0.21 * size);
    }
}
```

As you can see, in the `resize()` method, the only thing you have to do for each node is calculate its `size` and relocate the node. So first you calculate the minimum dimension of the LED (because it's square, you take the width if it's less than the height or vice versa). In addition, you must make sure that the resizing will be done only if the current size is larger than 0.

How It Works

Now that you have all things in place, you can simply use the control like any other control, as shown here:

```
@Override public void start(Stage stage) {  
    Led control = new Led();  
    StackPane pane = new StackPane();  
    pane.getChildren().add(control);  
    Scene scene = new Scene(pane);  
    stage.setTitle("JavaFX Led Control");  
    stage.setScene(scene);  
    stage.show();  
    control.setBlinking(true);  
}
```

Figure 15-17 shows the result.

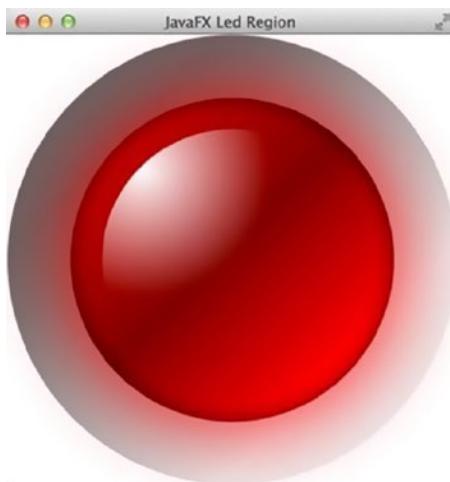


Figure 15-17. The region-based custom LED control

Other Ways to Create a Custom Control

As mentioned earlier, there are different ways to create a custom control in JavaFX. The “extend Region” approach that was shown here is the most common one. The drawback that comes with this approach is that you mixed up the model/controller with the view, which is acceptable for small controls like this LED, but might be a bad solution for a bigger control that relies on a data model or controls that are part of a controls library.

For those more complex controls, you should use a custom `Control.class` class in combination with a custom `Skin.class` and your CSS file. The main difference here is that the properties with their get, set, and property methods will be placed in the `Control.class` and the visualization code will be placed in the `Skin.class`. This approach provides proper separation between the controller logic and the view logic.

The last approach to creating a custom control that I want to mention is using the `Canvas` node. The `Canvas` node represents a single node that in principle behaves like an image that you can draw on. The API that is used to draw on the `Canvas` node is taken from the HTML5 `Canvas`, which means you won’t have JavaFX nodes but you will simply draw on the `Canvas` node. (That is, you’ll be working in immediate mode rather than retained mode.) You have to take care of clearing and redrawing the `Canvas` node surface. This approach might be useful for controls that contain very complex drawings to calculate and draw, as you would only need to do so once.

Summary

In this chapter, you learned how to style your application with custom CSS files by using the following methods:

- `Application.setUserAgentStylesheet(String URL)`
- `Scene: getStylesheets().add(String URL)`

Then you saw how CSS is used in JavaFX by using the default CSS selectors or creating your own. You learned about the `id` type selectors, `class` type selectors, and CSS *pseudo-classes*. In addition, you learned how to use the selectors in different selector patterns. By using selector patterns, you could apply styling to a group of nodes and their siblings.

Finally, you learned how to create an LED custom control in JavaFX. In this section, you got a chance to employ different strategies in creating custom controls. You learned how to use the `Control` and `Skin` classes to adhere to the standard way to create custom controls. You also learned about other interesting strategies, such as extending from `Region` or using the `Canvas` class to render primitives in immediate mode.

CHAPTER 16



Appendix A: References

Does your JavaFX journey stop here? Of course not! In this appendix, you will find many useful links and references that will help you gain further knowledge about all things JavaFX. Toward the end of this appendix, be sure to check out the many frameworks, libraries, and projects that use JavaFX in production today. In this edition I made an attempt to get feedback from the community based on useful resources related to JavaFX. In doing so, I may have missed some people and I want to apologize in advance. Many thanks to all of those who have contributed to the JavaFX community over the years!

Java 9 SDK

JavaFX 9 is built into Oracle's Java 9 SDK, which means that you only need to download the Java 9 SDK. The Java 9 software development kit and related information can be downloaded from the following locations.

- Java 9 at Oracle Technology Network:
<https://docs.oracle.com/javase/9/>
- Where to download the Java development kit:
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Java 9 API Documentation

The Java 9 documentation and guides are at the following links:

- Java 9 Javadoc API documentation:
<http://docs.oracle.com/javase/9/>
- JavaFX 9 Javadoc API documentation (subject to change):
<http://download.java.net/java/jdk9/jfxdocs/index.html?overview-summary.html>
<http://docs.oracle.com/en/java/>

Java 9 Features

Check out the following links for more specific and evolving information about Java 9:

- What is new in Java 9?
<https://docs.oracle.com/javase/9/whatsnew/toc.htm>
- 5 Features in Java 9 that Will Change How You Develop Software (and 2 That Won't), Alex Zhitnitsky, June 17, 2015 at Takipi Blog:
<http://blog.takipi.com/5-features-in-java-9-that-will-change-how-you-develop-software-and-2-that-wont/>

- Java 9 Features with Examples, Rambabu Posa, February 27, 2017:
<http://www.journaldev.com/13121/java-9-features-with-examples>
- Java 9 New Features, Eugen Baeldung:
<http://www.baeldung.com/new-java-9>

Java 9 Jigsaw

Check out the following links for more specific and evolving information about Java 9 Jigsaw:

- Project Jigsaw (OpenJDK):
<http://openjdk.java.net/projects/jigsaw>
- Quick Start Guide:
<http://openjdk.java.net/projects/jigsaw/quick-start>
- Migrating to Java 9:
<https://docs.oracle.com/javase/9/migrate/toc.htm>
- Java 9 modules big kill switch (--permit-illegal-access):
<http://mail.openjdk.java.net/pipermail/jigsaw-dev/2017-March/011763.html>
- The expert groups and industry leaders vote to move the Java 9 module system forward:
<https://jcp.org/en/jsr/results?id=6016>

IDEs

Integrated development environments are used to enter source code and compile Java code. The following are popular IDEs used in the industry.

- NetBeans: <https://netbeans.org/>
- IDEA IntelliJ: <http://www.jetbrains.com/idea>
- Eclipse: <https://www.eclipse.org/>
- BlueJ: <http://www.bluej.org/>

Deploying Applications

When deploying JavaFX applications, you'll want to use the following libraries and tools for packaging your applications.

- Deploying JavaFX Applications:
<https://docs.oracle.com/javase/9/deploy/toc.htm>
- Packaging JavaFX Applications as Native Installers:
<http://fxexperience.com/2012/03/packaging-javafx-applications-as-native>
- Launch4J:
<http://launch4j.sourceforge.net>
- FXLauncher:
<https://github.com/edvin/fxlauncher>

JavaFX 2D Shapes

The Javadoc API documentation includes abundant information about drawing shape nodes onto the JavaFX scene graph:

- <http://docs.oracle.com/javase/8/javafx/api/javafx.scene.shape/Shape.html>
- <http://docs.oracle.com/javase/8/javafx/api/javafx.scene.shape/Path.html>
- <http://docs.oracle.com/javase/8/javafx/api/javafx.scene.shape/SVGPath.html>
- <http://docs.oracle.com/javase/8/javafx/api/javafx.scene.text/Text.html>

JavaFX Color

The Javadoc API documentation also includes the following information about gradient color fills:

- <http://docs.oracle.com/javase/8/javafx/api/javafx.scene.paint/LinearGradient.html>
- <http://docs.oracle.com/javase/8/javafx/api/javafx.scene.paint/RadialGradient.html>
- <http://docs.oracle.com/javase/8/javafx/api/javafx.scene.effect/Blend.html>
- <http://docs.oracle.com/javase/8/javafx/api/javafx.scene.paint/ImagePattern.html>

JavaFX 2.x Builder Classes

Prior to JavaFX 8, the use of builder classes was quite widespread for very good reasons. Certain bugs, however, caused a conflict with the new Java 8, so the JavaFX teams decided to deprecate the builder classes. The following links discuss the change:

- A discussion on the Oracle JavaFX forum about the reason for deprecating builder classes: <https://community.oracle.com/thread/2544323?tstart=0>
- An e-mail thread from the Open JDK on the real reason why JavaFX Builder classes were deprecated in Java 8: <http://mail.openjdk.java.net/pipermail/openjfx-dev/2013-March/006725.html>

JavaFX Printing

New to Java 8 is the JavaFX printing API. This book does not discuss how to use the printing API, but the following links will help you get started:

- Javadoc API documentation:
<http://docs.oracle.com/javase/8/javafx/api/javafx/print/package-summary.html>
- *All the Nodes That Are Fit to Print (Part 1)*, Phil Race (Oracle):
<https://www.youtube.com/watch?v=Ma506QBmj90>
- *All the Nodes That Are Fit to Print (Part 2)*, Phil Race (Oracle):
<https://www.youtube.com/watch?v=-fE0t7L0v-8>

Project Lambda

The core Java 8 added language features are lambda expressions and the stream API. The following references are roadmaps, blogs, and videos of topics surrounding project lambda.

- State of the Lambda, Brian Goetz (Oracle):
<http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html>
- Q & A with Brian Goetz and Stuart Marks (NightHacking tour hosted by Stephen Chin from Oracle):
<https://www.youtube.com/watch?v=vaSv3FjUIVI>
- Java 8: Lambdas, Part 2, Ted Neward:
<http://www.oracle.com/technetwork/articles/java/architect-lambdas-part2-2081439.html>
- Java 8 Revealed: Lambdas, Default Methods, and Bulk Data Operations, Anton Arhipov:
<http://zeroturnaround.com/rebellabs/java-8-revealed-lambdas-default-methods-and-bulk-data-operations>
- 10 Examples of Lambda Expressions and Streams in Java 8, Javin Paul:
<http://javarevisited.blogspot.com/2014/02/10-example-of-lambda-expressions-in-java8.html>
- Java SE 8: Lambda Quick Start, Oracle:
<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html>
- Java 8: Lambda Expression Basics, Blue Sky Workshop:
<http://blueskyworkshop.com/topics/Java-Pages/lambda-expression-basics>
- Java 8: Closures, Lambda Expressions Demystified, Frank Hinkel:
<http://frankhinkel.blogspot.com/2012/11/java-8-closures-lambda-expressions.html>
- Maurice Naftalin's Lambda FAQ:
<http://www.lambdafaq.org>
- Tutorials on lambda expressions, Oracle:
<http://docs.oracle.com/javase/tutorial/java/java00/lambdaexpressions.html>
- Functional Programming in Java: Harnessing the Power Of Java 8 Lambda Expressions, Venkat Subramaniam. Pragmatic Bookshelf, 2014. ISBN-13: 978-1937785468.
- Java 8 Streams, Sven Ruppert (@SvenRuppert) (German version by Entwickler Press):
<http://www.amazon.de/dp/B00HALCBMC>
<http://www.barnesandnoble.com/w/java-8-streams-sven-ruppert/1117767060>

Nashorn

Java 8 includes a new scripting engine called Nashorn, which is a new and improved JavaScript engine for the Java runtime. The engine enables developers to use JavaScript to program applications. The following links and references are articles and blogs describing Nashorn.

- Nashorn and JavaFX:
<https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/nashorn/javafx.html>
- Oracle's Nashorn: A next Generation JavaScript Engine for the JVM, Julien Ponge:
<http://www.oracle.com/technetwork/articles/java/jf14-nashorn-2126515.html>
- Open JDK's Nashorn site:
<http://openjdk.java.net/projects/nashorn/>
- The Nashorn blog:
<https://blogs.oracle.com/nashorn>

Properties and Bindings

Properties and bindings are essential to JavaFX when synchronizing values between JavaFX nodes. The following are great resources about read-only properties, listeners, and the role of JavaFX Beans.

- Creating Read-Only Properties in JavaFX, Michael Heinrichs:
<http://blog.netopyr.com/2012/02/02/creating-read-only-properties-in-javafx>
- When to Use a ChangeListener or an InvalidatedListener, Michael Heinrichs:
<http://blog.netopyr.com/2012/02/08/when-to-use-a-changelistener-or-an-invalidatedlistener>
- The Unknown JavaBean, Richard Bair:
<https://community.oracle.com/blogs/rbair/2006/05/31/unknown-javabean>
- Using JavaFX Properties and Binding, Scott Hommel:
<http://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm>
- Creating JavaFX Properties, Michael Heinrichs:
<http://blog.netopyr.com/2011/05/19/creating-javafx-properties>
- Pro JavaFX 8, (Chapter 4, “Properties and Bindings”), Johan Vos, James Weaver, Weiqi Gao, Stephen Chin, and Dean Iverson, Apress, 2014:
<http://www.apress.com/9781430265740>
- mvvm(fx): An application framework that provides the necessary components to implement the MVVM pattern with JavaFX:
<https://github.com/sialcasa/mvvmFX>
- Open Dolphin: A JavaFX MVC Framework (founded by Dierk Koenig of Canoo Engineering):
http://open-dolphin.org/dolphin_website/Home.html
- JavaFX MVP Framework Based on Convention over Configuration and Dependency Injection (founded by Adam Bien):
<http://afterburner.adam-bien.com>

Layouts

Building UIs can be a challenge when it comes to layout. The following references tutorials and libraries discuss how to use JavaFX nodes.

- Laying Out a User Interface with JavaFX 2.0, Jim Weaver (Oracle):
<http://www.oracle.com/technetwork/articles/java/layoutfx-1536156.html>
- Working With Layouts in JavaFX, Joni Gordon:
<http://docs.oracle.com/javafx/2/layout/jfxpub-layout.htm>
- Pro JavaFX 2, Chapter 4, “Building Dynamic UI Layouts in JavaFX” (Apress, 2012):
<http://www.apress.com/9781430268727>
- In-Depth Layout and Styling with the JavaFX Scene Builder:
http://www.youtube.com/watch?v=7Nu3_5doZK4
- Miglayout, Mikael Grev: <http://www.miglayout.com>
- JavaFX 2.0 EA and MigLayout, Tom Eugelink:
<http://tbeernot.wordpress.com/2011/03/11/javafx-2-0-ea-and-miglayout>
- Flex Box FX: Responsive Design for JavaFX. JavaFX Port of CSS3 FlexBox Layout Manager:
<http://flexboxfx.io>
- JFoenix: A Material Design Framework for JavaFX:
<http://www.jfoenix.com>
- BootstrapFX: Bootstrap Port for JavaFX:
<https://github.com/aalmiray/bootstrapfx>
- Jbootx: Bootstrap Look for JavaFX:
<https://github.com/dicolar/jbootx>

JavaFX Tools

You'll find helpful tools from building GUIs to debugging applications in the following links:

- Scene Builder: A tool to build JavaFX-based UIs using the FXML markup language:
<http://www.oracle.com/technetwork/java/javafx/tools/index.html>
- Scenic View tool: A Tool to Visually Debug JavaFX UIs during Runtime:
<http://fxexperience.com/scenic-view>
- Caspian Styler, Animation Spline Editor and Derived Color Calculator:
<http://fxexperience.com/2012/03/announcing-fx-experience-tools>
- JavaFX Tooling and Runtime for Eclipse and OSGi (founder Tom Schindl):
<http://www.eclipse.org/efxclipse/index.html>
<http://tomsondev.bestsolution.at>
- Firebug Lite: A Browser-Based Debugger Tool for JavaFX WebView Node:
<http://stackoverflow.com/questions/17387981/javafx-webview-webengine-firebuglite-or-some-other-debugger>
- JavaFX 3D Viewer and Importer:
<http://www.interactivemesh.org/models/jfx3dimporter.html>

- IDR Solutions Showcasing a PDF Viewer for JavaFX:
<http://blog.idrsolutions.com/2014/03/sneak-preview-new-javafx-pdf-viewer>
- TestFX: Easy-to-Use Library for Testing JavaFX:
<https://github.com/TestFX/TestFX>
- Introducing MarvinFx, Hendrik Ebbers: Easily Tests JavaFX Controls and Scenes with a Special Attention to Properties:
<http://www.guigarage.com/2013/03/introducing-marvinfox>
- JemmyFX: Provides an API for Testing JavaFX User Interface in Your Application:
<https://jemmy.java.net/JemmyFXGuide/jemmy-guide.html>
- ReactiveX/RxJavaFX by Thomas Nield: JavaFX bindings for RxJava. RxJavaFX is a lightweight library to convert JavaFX events into RxJava Observables/Flowables and vice versa. It also has a Scheduler to safely move emissions to the JavaFX Event Dispatch Thread:
<https://github.com/ReactiveX/RxJavaFX>
Twitter: @thomasnield9727

Enterprise GUI Frameworks

GUI frameworks enable developers to develop enterprise applications rapidly. From binding to MVC framework patterns, the following will help you become productive when developing JavaFX applications.

- Gluon Mobile: Gluon provides an end-to-end enterprise mobile solution for developing cross-platform mobile apps that easily connect to enterprise backends and cloud services, all while being centrally managed:
<http://gluonhq.com>
- Griffon (project lead Andres Almiray, [@aalmiray](https://twitter.com/aalmiray)): Griffon is an application framework for developing desktop applications in the JVM:
<http://griffon-framework.org>
- jpro: A new technology that brings Java back into the browser without a Java plugin. To achieve that, jpro runs JavaFX on the server and maps its scene graph directly into the browser. The client-side rendering is highly optimized with browser side approximations to get a smooth user experience free of lags:
<https://www.jpro.io>
Twitter: https://twitter.com/jpro_io
Co-founder Tobias Bley, [@tobibley](https://twitter.com/tobibley)
- DukeScript: A new technology for creating cross-platform mobile, desktop, and web applications. DukeScript applications are plain Java applications that internally use HTML5 technologies and JavaScript for rendering. This way, developers only need to write clean Java code and can still leverage the latest developments in modern UI technology:
<https://dukescrpt.com>
Twitter: @DukeScript
- TornadoFX by Edvin Syse: Lightweight JavaFX Framework for the Kotlin language:
GitHub: <https://github.com/edvin/tornadofx>
Twitter: @edvinsyse
- Open Dolphin: A JavaFX MVC Framework (founded by Dierk Koenig (@mittie) of Canoo Engineering):
http://open-dolphin.org/dolphin_website/Home.html

- JavaFX MVP framework based on convention over configuration and dependency injection. Founded by Adam Bien (@AdamBien):
<http://afterburner.adam-bien.com>
- ReduxFX 0.2 by Michael Heinrichs:
<https://netopyr.com/2017/02/27/new-features-reduxfx-0-2/>
<https://netopyr.com>
Twitter: @netopyr
- NetBeansIDE-AfterburnerFX-Plugin by Peter Rogge: The NetBeansIDE-AfterburnerFX-Plugin is a NetBeans IDE plugin that supports the file generation in convention with the library `afterburner.fx` in a JavaFX project:
<https://github.com/Naoghuman/NetBeansIDE-AfterburnerFX-Plugin>
Twitter: @naoghuman
- FXForm2: A library providing automatic JavaFX 2.0 form generation:
<http://dooapp.github.io/FXForm2>
- Integrating Spring Boot with JavaFX by MVP Java:
<https://youtu.be/hjeS0xi3uPg>
- JavaFX Multiple Controllers by MVP Java:
<https://www.youtube.com/watch?v=osIRfgHTfyg>

Domain-Specific Languages

A domain-specific language for graphics can help eliminate a lot of boilerplate code, while keeping code easier to read and manage. The following are DSLs for JavaFX in various computer languages that run on top of the Java runtime environment.

- TornadoFX: A Kotlin-based domain-specific language to create JavaFX applications:
<https://github.com/edvin/tornadofx>
- GroovyFX: A Groovy-based domain-specific language to create JavaFX applications:
<http://groovyfx.org>
- ScalaFX: A Scala-based domain-specific language to create JavaFX applications:
<https://code.google.com/p/scalafx>
- RubyFX: A Ruby-based domain-specific language to create JavaFX applications:
<https://github.com/jruby/jrubyfx>
- Visage: A JavaFX-based domain-specific language to create JavaFX applications:
<https://code.google.com/p/visage>

Custom UIs

You can customize a user interface such as styling using CSS or using custom controls. The following links are reference guides and blogs.

- JavaFX 8 CSS reference:
<http://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html>
- JavaFX FXML introduction:
http://docs.oracle.com/javase/8/javafx/api/javafx/fxml/doc-files/introduction_to_fxml.html

- Styling UI Controls with CSS:
<http://docs.oracle.com/javase/8/javafx/user-interface-tutorial/apply-css.htm>
- *Modena Theme for JavaFX 8*, Jasper Potts:
<http://fxexperience.com/2013/03/modena-theme-update>
- *Mastering JavaFX 8 Controls*, Hendrik Ebbers. Oracle Press, 2014.
ISBN 9780071833776:
<http://mhprofessional.com/product.php?isbn=0071833773>
<http://www.guigarage.com/javafx-book>
- A MacOS X look and feel called AquaFX with Elements, Claudine Zillmann (@etteClaudette):
<http://aquaafx-project.com/project.html>
- A Windows 8 look and feel called JMetro, Pedro Duque Vieira (@P_Duke):
<http://pixelduke.wordpress.com/category/metro>
<https://github.com/JFXtras/jfxtras-styles>
- Orson Charts by David Gilbert: A 3D chart library for Java applications (JavaFX, Swing, or server-side).
<https://github.com/jfree/orson-charts>
- Roku-like or Apple TV-like interface called the Flatter look and feel, Hendrik Ebbers (@hendrikEbbers):
<http://www.guigarage.com/2013/09/flatter>
<http://www.guigarage.com/2013/08/boxfx-javaone-preview-1>
- GuiGarage's collection of JavaFX talks at JavaOne (provided by Parleys):
<http://www.guigarage.com/2013/12/javafx-at-javaone>
- *Why Use FXML* by Irina Fedortsova of Oracle:
http://docs.oracle.com/javafx/2/fxml/get_started/why_use_fxml.htm
- Gerrit Grunwald's JavaFX 8 examples using many different strategies for making custom controls:
<https://github.com/HanSolo/JFX8CustomControls>
- Enzo: JavaFX gauges, clocks, and many custom controls, Gerrit Grunwald:
<https://bitbucket.org/hansolo/enzo/wiki/Home>
- Medusa by Gerrit Grunwald: JavaFX Custom Controls library to create Gauges and Dials (non CSS).
<https://bintray.com/hansolo/Medusa/Medusa>
Twitter: @hansolo_
- JFXtras: A JavaFX custom controls community:
<http://jfxtras.org>
- ControlsFX: Another custom controls community started and lead, Jonathan Giles of Oracle:
<http://fxexperience.com/controlsfx>
- *JideFX Highlights of the JideFX Beta Release (1 of 3)*, David Qiao:
<http://www.jidesoft.com/blog/2013/06/10/highlights-of-the-jidefx-beta-release-1-of-3>
<http://www.jidesoft.com>

- *Styling FX Buttons with CSS*, Jasper Potts:
<http://fxexperience.com/2011/12/styling-fx-buttons-with-css>
- *JavaFX Custom Control: Nest Thermostat Part 1-3*, Laurent Nicolas:
<http://www.javacodegeeks.com/2014/01/javafx-custom-control-nest-thermostat-part-1.html>
Twitter: @MrLoNee
- *Do It Yourself: Custom JavaFX Controls*, Gerrit Grunwald:
https://www.youtube.com/watch?v=ts_b2mBix3U
- Custom Controls by SIB Visions: FXSelectableLabel, FXMonthView, FXDatePicker, and many more:
<https://blog.sibvisions.com/2015/04/24/javafx-custom-controls/>
Twitter: @sibvisions
- yFiles for JavaFX by yWorks GmbH: A library that brings the proven power and ease of yFiles diagramming to your cutting-edge JavaFX applications. The library contains UI controls for drawing, viewing, and editing diagrams and mature graph layout algorithms for automatically arranging complex graphs and networks at the click of a button:
<http://www.yworks.com/products/yfiles-for-javafx>
- SyntheticaFX made by JylooSoftware: The Library for JavaFX Business/Enterprise Solutions. Provides themes and components mainly made for professional business applications on the desktop. The library is growing and new controls are under construction and will be added in future releases. The target platform of the final release is Java 9 or above:
<http://www.jyloo.com/syntheticafx/>
Founder: W.Zitzelsberger
Twitter: @wzberger
- JideFX by Jide Software, Inc.: A collection of various extensions and utilities for the JavaFX platform. The JideFX Common Layer is the equivalent to the JIDE Common Layer in the JIDE components for Swing:
<https://github.com/jidesoft/jidefx-oss>
- DateAxis supporting JSR-310 by Pedro Duque Vieira: JavaFX Charts using XYBarChart can now take advantage of the new Date APIs in Java 8:
<https://pixelduke.wordpress.com/2013/12/13/dateaxis-and-xybarchart-update/>
- FXValidation by Pedro Duque Vieira: A JavaFX validation library to work with FXML GUI forms:
<https://pixelduke.wordpress.com/2014/07/26/validation-in-java-javafx/>
GitHub: <https://github.com/dukke/FXValidation>
Twitter: @P_Duke
- FXRibbon by Pedro Duque Vieira: A GUI control common in Microsoft Office applications:
Blog: <https://pixelduke.wordpress.com/2015/01/11/ribbon-for-java-using-javafx/>
GitHub: <https://github.com/dukke/FXRibbon>
- Custom JavaFX Styling by Narayan G. Maharjan:
<http://blog.ngopal.com.np/2012/07/11/customize-scrollbar-via-css/>
<http://blog.ngopal.com.np>
Twitter: @javadeveloping

- Undecorator by Arnaud Nouard: A library to create irregular shaped semi-transparent windows for your applications: <https://github.com/in-sideFX/UndecoratorBis>
Twitter: @In_SideFX
- JavaFx JFoenix Tutorial #2 : Material Design Buttons:
<https://youtu.be/22Ql0j6JVe4>
- Ten Ways to Make Your JavaFX App Shine by Martin Gunnarsson and Pär Sikö:
<https://www.youtube.com/watch?v=98ZmnN215m0>

Operating System Style Guidelines

In JavaFX there are many ways to develop esthetically pleasing applications; however, there are times you will need style guidance on the host operating system. The following links are many of the mainstream operating system style guidelines.

- iOS7 style guidelines:
<https://developer.apple.com/library/ios/documentation/userexperience/conceptual/MobileHIG/index.html>
- Android style guidelines:
<http://developer.android.com/design/style/index.html>
- Windows style guidelines:
<http://msdn.microsoft.com/en-us/library/windows/apps/hh465424.aspx>
- Ubuntu design guidelines:
<http://design.ubuntu.com/apps>
- BlackBerry design guidelines:
http://developer.blackberry.com/native/documentation/cascades/best_practices/uiguidelines
- *HTML Styles: CSS* by w3schools.com:
http://www.w3schools.com/html/html_css.asp

JavaFX Media

JavaFX contains a powerful media API capable to process audio and video. Here you will find many links that assist you in taking advantage of the `MediaView` and `MediaPlayer` APIs. Also, you will be seeing many links that help you create content.

- Oracle's tutorial to incorporating media using JavaFX:
<http://docs.oracle.com/javase/8/javafx/media-tutorial/overview.htm>
- *Audio and Video Processing in JavaFX* by Brian Burkhalter and David DeHaven (principal members of the technical staff at Oracle):
<https://www.youtube.com/watch?v=jaPUbfJx2A>
- *Pro JavaFX 2*, Chapter 8, “Media Classes” (Apress, 2014):
<https://www.apress.com/9781430268727>
- Media College:
<http://www.mediacollege.com/adobe/flash/video/tutorial/example-flv.html>
- VP6 encoder for FLV container formats:
<http://www.wildform.com>

- Sony Vegas:
<http://www.sonycreativesoftware.com/vegassoftware>
- Adobe Premier:
<http://www.adobe.com/products/premiere.html>
- Apple's Final Cut Pro: Movie making software:
<http://www.apple.com/final-cut-pro>
- Apple's iMovie: Movie making software:
<https://www.apple.com/mac/imovie>
- Apple's Quick Time Pro: A media player:
<http://www.apple.com/quicktime>
- FFmpeg: a complete, cross-platform solution to record, convert, and stream audio and video:
<http://www.ffmpeg.org>
- On2 Technologies is acquired by Google:
http://en.wikipedia.org/wiki/On2_Technologies

JavaFX on the Web

Many of the following links relate to the powerful WebView node in JavaFX and how to interact with web services. Also, you will see some new and interesting concepts such as JavaFX in the browsers using JavaScript.

- Overview of the JavaFX WebView component:
<http://docs.oracle.com/javase/8/javafx/embedded-browser-tutorial/overview.htm>
- A talk on HTML5 and JavaFX by Ryan Cuprak (at the Silicon Valley JavaFX User Group):
<https://www.youtube.com/watch?v=xgdD7EBX4fw>
- Other w3c document object model APIs: JDOM:
<http://www.jdom.org>
- DOM4J:
<http://dom4j.sourceforge.net>
- Bug RT-35757 relating to loading files containing HTML5 with a file extension other than .html:
<https://javafx-jira.kenai.com/browse/RT-35757>
- Converting from w3c documents to a string:
<http://stackoverflow.com/questions/2567416/document-to-string>
- Loading a single text file as a string, by Pat Niemeyer:
https://weblogs.java.net/blog/pat/archive/2004/10/stupid_scanner.html
- Single-line read text file with Java by Fuad Saud:
<https://coderwall.com/p/lyalwg>
- How to read data from InputStream into String in Java by Lokesh Gupta:
<http://howtodoinjava.com/2013/10/06/how-to-read-data-from-inputstream-into-string-in-java>

- Java 8 Streams API to read a file by Cay Horstmann:
<https://weblogs.java.net/blog/cayhorstmann/archive/2013/12/06/java-8-really-impatient>
- Firebug Lite:
<http://stackoverflow.com/questions/17387981/javafx-webview-webengine-firebuglite-or-some-other-debugger/18396900>
- Reading JSON using URL Connection:
<http://docs.oracle.com/javase/tutorial/networking/urls/readingWriting.html>
- Making a POST request instead of a GET using URL Connection:
<http://stackoverflow.com/questions/4205980/java-sending-http-parameters-via-post-method-easily>
- JavaScript 12 hour AM and PM format dates:
<http://stackoverflow.com/questions/8888491/how-do-you-display-javascript-datetime-in-12-hour-am-pm-format>
- OpenWeatherMap: free weather data and forecast API:
<http://openweathermap.org>
- Free Geo-location:
<http://freegeoip.net/json>
- JSObject Live connect:
https://developer.mozilla.org/en-US/docs/LiveConnect/LiveConnect_Reference/JSObject
- DataFX, Johan Vos, Jonathan Giles, and Hendrik Ebbers:
<http://www.javafxdata.org>
- *Building Rich Client Applications with JSR 356: Java API for WebSocket*, Johan Vos:
http://www.oraclejavamagazine-digital.com/javamagazine_twitter/20140102#pg67
- Bck2Brwsr: JavaFX in the browser:
<http://wiki.apidesign.org/wiki/Bck2Brwsr>
- Canvas for Bck2Brwsr:
<http://jayskills.com/blog/2013/01/22/canvas-for-bck2brwsr>

JavaFX 3D

The highly anticipated JavaFX 3D API has finally arrived, and it is part of the Java 8 release. The following links are to videos and tutorials that will enrich your JavaFX 3D skills:

- JavaFX 3D Oracle tutorial:
http://docs.oracle.com/javafx/8/3d_graphics/jfxpub-3d_graphics.htm
<http://docs.oracle.com/javase/8/javafx/graphics-tutorial/javafx-3d-graphics.htm>
- JavaFX for Business: Monitoring a Container Terminal at JavaOne 2012, Dierk Konig and Andres Almiray:
<https://www.youtube.com/watch?v=m5NcuWGKnDo>

- Modeling, Texturing, and Lighting Mesh Geometry in JavaFX 3D, John Yoon of Oracle:
<http://www.parleys.com/share.html#play/5252e81de4b0c4f11ec576b0>
- JavaFX 3D: The Third Dimension [CON7810] at JavaOne 2013, Kevin Rushforth and Chien Yang:
https://oracleus.activeevents.com/2013/connect/sessionDetail.ww?SESSION_ID=7810
- Polaris, JavaFX, ControlsFX, and F(X)yz supporting NASA Missions, Sean Phillips:
<https://www.youtube.com/watch?v=aA5GfwC8ho4>

JavaFX Gaming

And you thought JavaFX was only for business. Well, take a look at the following links to see some fun and interesting games.

- JavaFX Game Development Framework:
<https://github.com/AlmasB/FXGL>
Twitter: @AlmasB
- JavaFX Games and OpenCV App examples using Scala, Robert Ladstätter:
<https://github.com/rladstaetter>
Twitter: @rladstaetter
- GameComposer, Mirko Sertic: A game authoring tool and a game runtime environment targeting desktop and mobile devices:
GitHub: <https://github.com/mirkosertic/GameComposer>
Twitter: @mirkosertic
- *Writing a Tile Engine in JavaFX*, Toni Epple:
<http://www.javacodegeeks.com/2013/01/writing-a-tile-engine-in-javafx.html>
- SokubanFX, Peter Rogge: A JavaFX-based game that is a type of transport puzzle, in which the player pushes boxes or crates around in a warehouse, trying to get them to storage locations. The puzzle is usually implemented as a video game:
<https://github.com/Naoghuman/SokubanFX>
Twitter: @naoghuman
- *Tower Defense type game by RolandC*:
<http://wecode4fun.blogspot.co.at/2015/05/simple-game-engine-tower-defense.html>
- Angry Duke: Physics-Based Games in JavaFX, Toni Epple:
<https://www.youtube.com/watch?v=jcSp2EoMTyE>
- JavaOne 2013 Chess Robot, Jasper Potts: In the JavaOne 2013 technical keynote, the Java team demonstrated a robot controlled by Java playing chess:
<http://vimeo.com/75368008>
- A JavaFX based Game Authoring System, Mirko Sertic:
<http://www.mirkosertic.de/doku.php/javastuff/javafxgameauthoring>
- Rubik's Cube with JavaFX 3D II, José Pereda:
<https://www.youtube.com/watch?v=ZVPIBkDgZV4>
<https://github.com/jperedadnr/RubikFX>

- A JavaFX game tutorial Parts 1-6, Carl Dea:
<http://carlfx.wordpress.com/2012/03/29/javafx-2-gametutorial-part-1>

Java IoT and JavaFX Embedded

The following links and references are very cool DIY projects using the combination of Java embedded and JavaFX. Many of these links are videos that will help you see a glimpse into the future of the Internet of Things.

- Java 8 Embedded documentation:
<http://docs.oracle.com/javase/8/javase-embedded.htm>
- Robo4J by Miro Wengner and Marcus Hirt: A framework for quickly getting started building and running robots and IoT devices:
<http://www.robo4j.io/p/home.html>
Video: <http://www.robo4j.io/p/videos.html>
Miro Wengner: @miragemiko
Marcus Hirt: @hirt
- PiDome: Home automation PiDome is an Open Source Home Automation platform developed especially for the Raspberry Pi, leveraging it to a full home automation solution:
<https://pidome.org>
Twitter: @Pi_Dome
- MQTT.fx by Jens Deter: The JavaFX based MQTT Client:
<http://www.mqttfx.org>
Twitter: @MQTT_fx
Twitter: @Jerady
<http://www.jensd.de>
- JavaFX-based tablet computer:
<http://j.mp/DukePad>
<https://wiki.openjdk.java.net/display/OpenJFX/DukePad>
<http://vimeo.com/76486818>
- Nao Robot NightHacking:
<http://nighthacking.com/nao-robot-nighthacking-part-1>
<http://nighthacking.com/nao-robot-nighthacking-part-2>
- Pre-interview with Mark Heckler (NightHacking tour hosted, Stephen Chin of Oracle):
<https://www.youtube.com/watch?v=LXRp57B58e8>
- NightHacking with Gerrit Grunwald on Pi, XBee, and Sensor Networks. (NightHacking tour hosted, Stephen Chin of Oracle):
<https://www.youtube.com/watch?v=GTroCtrlNGY>
- NightHacking with Gerrit Grunwald and Michael Hoffer. (NightHacking tour hosted by Stephen Chin of Oracle):
<https://www.youtube.com/watch?v=cWFW4VT8auw>
- *Beyond Beauty: JavaFX, Parallax, Touch, Gyroscopes and Much More*, Angela Caicedo (Java Evangelist at Oracle) (NightHacking tour hosted by Stephen Chin of Oracle):
<https://www.youtube.com/watch?v=rYt45rs2Bjk>
- Raspberry Pi Java forum:
<http://www.raspberrypi.org/phpBB3/viewforum.php?f=81>

- How to safely overclock your Raspberry Pi, Eben Upton (Raspberry Pi Founder, CEO of Raspberry Pi's engineering team):
<http://www.raspberrypi.org/archives/2008>
- End-to-End Raspberry Pi with NetBeans IDE 8, José Pereda:
<http://netbeans.dzone.com/articles/nb-8-raspberry-pi-end2end>
Twitter: @JPeredaDnr

Software and Device Manufacturers

In the last few chapters we discussed Java embedded devices and gesture-based input devices. The following links reference various manufacturers.

- Raspberry Pi: <http://www.raspberrypi.org>
- Arduino: <http://www.arduino.cc>
- BeagleBoard: <http://beagleboard.org/Products/BeagleBone>
- Leap Motion: <https://www.leapmotion.com>
- Microsoft Windows 8 Surface: <http://www.microsoft.com/surface/en-us>
- Oculus Rift:
<http://www.oculusvr.com>
- Thalmic Labs:
<https://www.thalmic.com/en/myo>

JavaFX Communities

So you want to get involved with the JavaFX community? Please check out the following links.

- Java.net JavaFX community site:
<https://www.java.net/community/javafx>
- FXExperience: JavaFX News, Demos and Insight:
<http://fxexperience.com>
Twitter: @fxexperience
- Nighthacking: Hosted by Stephen Chin. A tour around the world to see everything about Java, JavaFX, and IoT. Amazing live talks.
<http://nighthacking.com>
Twitter: @_nighthacking
- Oracle's JavaFX Community portal to Real World Use Cases, Community Support, Third Party tools and Open JFX:
<http://www.oracle.com/technetwork/java/javase/community/index.html>
- JFXtras: A JavaFX custom controls community:
<http://jfxtras.org>
- ControlsFX: Another custom controls community, started by Jonathan Giles of Oracle:
<http://fxexperience.com/controlsfx>
- Enzo by Gerrit Grunwald: JavaFX Custom Controls library to create Gauges and Dials using JavaFX CSS styling:
https://bintray.com/hansolo/Enzo/Enzo_All_In_One

- Feed Cluster: Any and all JavaFX news:
<http://javafx-blogs.feedcluster.com>
- Silicon valley JavaFX user's group:
<http://www.meetup.com/svjugfx>
- Silicon valley JavaFX user's group Live stream:
<http://www.ustream.tv/channel/silicon-valley-javafx-user-group>
- Oracle Forums on JavaFX:
https://community.oracle.com/community/developer/english/java/javafx/javafx_2.0_and_later

Applications

Many of the following links are rich GUI applications written using the JavaFX platform.

- 20 JavaFX real-world applications, Dirk Lemmermann and Alexander Casall:
<https://jaxenter.com/20-javafx-real-world-applications-123653.html>
Dirk Lemmermann: @dlemmermann
Alexander Casall: @sialcasa
- FXDesktopSearch, Mirko Sertic: A Java and JavaFX based desktop search application. It crawls a configured set of directories and allows you to do full-text searches with different language support:
<https://github.com/mirkosertic/FXDesktopSearch>
Twitter: @mirkosertic
- Picmodo (founded by Jens Deter,): A JavaFX-based picture database application:
http://www.jensd.de/wordpress/?page_id=22
Twitter: @Jerady
- VWorkflows by Michael Hoffer: A Scientific Visualization Tool:
<https://github.com/mihohsoft/VWorkflows>:
<http://mihosoft.eu>
Twitter: @mihosoft
- eteoBoard: The interactive digital task board. A touch and gesture-based application tool for team collaboration:
<http://www.eteoboard.de>
<http://www.saxsys.de>
- PiDome client, John Sirach: PiDome is an open source home automation platform developed for the Raspberry Pi leveraging it to a full home automation solution.
<https://pidome.org>
Twitter: @Pi_Dome
- FlexGanttFX: Aircraft/Flight Scheduling Application, by Dirk Lemmermann, @dlemmermann: <http://www.dlsc.com>
https://www.youtube.com/channel/UCFyRQ_euxxPDwlqyhff-x0Q
- BaufiFX, Thomas Bolz: A home finance calculator:
<http://www.baufifx.de>

- SnapCode is an IDE and RAD tool for education, entertainment, and the enterprise. Report Mill Software:
<http://www.reportmill.com/snap>
- Modellus is a freely available application that enables students and teachers (high school and university) to use mathematics to create or explore models interactively. Modellus was developed by Vitor Duarte Teodoro, João Paulo Duque Vieira, and Pedro Duque Vieira:
<http://modellus.co/index.php/en>
- MIME Brower by Christoph Nahr is a compact JavaFX desktop application for browsing MIME messages that are locally stored in standard EML files:
<http://kynosarges.org/MimeBrowser.html>
- Vocab Hunter by Adam Carroll: VocabHunter is a system to help learners of foreign languages:
GitHub: <https://vocabhunter.github.io>
Twitter: @vocabhunterapp
- Bloom Uploader by Carl Antaki. Bloom is a desktop app that lets you upload your photos and videos easily and efficiently to Facebook, download albums, view your photos and some of your friends' photos:
Video: <http://bloomuploader.com>
Twitter: @carcour
- AsciidocFX by Rahman USTA: Asciidoc FX is a book/document editor used to build PDF, Epub, Mobi, and HTML books, documents, and slides. AsciidocFX is also a winner of Duke's Choice Award 2015:
<https://github.com/asciidocfx/AsciidocFX>
Twitter: @ustarahman
- JITWatch Code Cache Visualization by Chris Newland. Visualize how the HotSpot JVM code cache fills up as the JIT compilers produce optimized native code from your bytecode:
<https://github.com/AdoptOpenJDK/jitwatch>
- MuseuID, Angelica Leite. The software assists the Museum of Paleontology in Santana do Cariri. It manages the collection of fossils and museum activities:
Video: <http://t.co/3MuofK4eZF>
GitHub: <https://github.com/angelicalalleite/museuid>
Twitter: @angelicalalleite
- AZoi Capital Atlas Trader Software JavaFX, Rob Terpilowski:
Video: <https://youtu.be/tm05AIkvJAs>
Twitter: @RobTerpilowski
- Power Scene Viewer 2.0, Marlon Trujillo: SCADA software:
<https://www.youtube.com/watch?v=8DUAf9TrJuI>
<https://www.youtube.com/watch?v=FPhph2jrSOEU>
Twitter: @marlon_marlon

- FXDesktopSearch: A Java and JavaFX-based desktop search application. It crawls a configured set of directories and allows you to do full-text searches with different language support:
<https://github.com/mirkosertic/FXDesktopSearch>

Java/JavaFX Books and Magazines

The following links are newer book titles that relate to the new Java 8 and JavaFX 8 platform.

- *Pro JavaFX 8*, Johan Vos, James Weaver, Weiqi Gao, Stephen Chin, Dean Iverson. Apress, 2014. ISBN: 978-1-4302-6574-0:
<http://www.apress.com/9781430265740>
- *Java 8 Recipes*, Josh Juneau (Apress, 2014. ISBN: 978-1-4302-6827-7):
<http://www.apress.com/9781430268277>
- *JavaFX Rich Client Programming on the NetBeans Platform*, Paul Anderson and Gail Anderson. Addison-Wesley Professional, 2014. ISBN: 978-0321927712:
https://blogs.oracle.com/geertjan/entry/new_book_javafx_rich_client
<http://www.amazon.com/JavaFX-Client-Programming-NetBeans-Platform/dp/0321927710>
- *Mastering JavaFX 8 Controls*, Hendrik Ebbers. Oracle Press, 2014. ISBN: 9780071833776:
<http://mhprofessional.com/product.php?isbn=0071833773>
<http://www.guigarage.com/javafx-book>
- *Quick Start Guide to JavaFX*, J.F. DiMarzio. Oracle Press, 2014. ISBN: 978-0071808965:
<http://www.mhprofessional.com/product.php?isbn=0071808965>
- *Java SE 8 for the Really Impatient*, Cay S. Horstmann. Addison-Wesley, 2014. ISBN: 978-0321927767:
<https://www.pearson.com/us/higher-education/program/Horstmann-Java-SE-8-for-the-Really-Impatient-A-Short-Course-on-the-Basics/PGM214606.html>
- *Mastering Lambdas*, Maurice Naftalin. Oracle Press, 2014. ISBN: 007-1829628:
<https://www.amazon.com/Mastering-Lambdas-Programming-Multicore-Oracle/dp/0071829628>
- *Java Magazine* from Oracle:
<http://www.oracle.com/technetwork/java/javamagazine/index.html>
- *JavaFX RefCard*, Hendrik Ebbers and Michael Heinrichs:
<https://dzone.com/refcardz/javafx-8-1>
- Java Everywhere, Anton Epple. A book on the DukeScript platform:
<https://leanpub.com/dukescript>
Twitter: @monacoton

Author Blogs

If you are having trouble with this book's content or code examples, feel free to comment or mention issues at the book's catalog page at apress.com, in the errata tab. If you are totally stumped, feel free to contact the book authors via Twitter or their blogs.

The following are the book's authors with their Twitter accounts and blog URLs:

- Carl Dea (@carldea): <http://carlfx.wordpress.com>
- Gerrit Grunwald (@hansolo_): <http://harmoniccode.blogspot.com>
- Mark Heckler (@MkHeck): <http://www.thehecklers.org/>
- José Pereda (@JPeredaDnr): <http://jperedadnr.blogspot.com>
- Sean Phillips (@SeanMiPhillips): <http://birdasaur.tumblr.com>

Tutorials, Courses, Consulting Firms, and Demos

The following links are where you will meet experts and enthusiasts alike who provide many free online tutorials and offer courses. Some of the links are presentations from major consulting firms teaching and demoing real-world applications.

- FXDocs by the JavaFX community: The JavaFX Documentation Project aims to pull together useful information for JavaFX developers from all over the web. The project is open source and encourages community participation to ensure that the documentation is as highly polished and useful as possible:
<https://fxdocs.github.io/docs/index.html>
- AwesomeJavaFX, Hossein Rimaz: A curated list of awesome JavaFX libraries, books, frameworks, etc.:
<https://github.com/mhrimaz/AwesomeJavaFX>
Twitter: @mhrimaz
- Various algorithms using JavaFX, Hossein Rimaz:
<https://github.com/mhrimaz>
Twitter: @mhrimaz
- JavaFX Training Courses with Paul and Gail Anderson, authors, teachers, and speakers:
Twitter: @Paul_ASGTeach and @gail_asgteach
<http://asgteach.com/courses/java-and-javafx/javafx-programming-course/>
- Bekwam Courses, Carl Walker. Tutorials and examples of building real-world JavaFX-based applications:
<https://www.bekwam.com>
https://courses.bekwam.net/public_tutorials/index.html
Twitter: @bekwaminc
- Excellent JavaFX Tutorials, Jörn Hameister:
<http://www.hameister.org/JavaFX.html>
Twitter: @Hameiste
- JavaFX Demos such as FXCubes and FXFrostedPanel, Alessio Vinerbi. Amazing JavaFX demos that shows a mix of video and 3D effects:
<https://github.com/alexbodogit>
Twitter: @Alessio_Vinerbi

- JavaFX tutorials, Jakob Jenkov:
<http://tutorials.jenkov.com/javafx/index.html>
Twitter:@jenkov
- JavaFX tutorials, Dirk Lemmermann Consulting. Excellent tutorials:
<http://dlsc.com/blog/>
Twitter:@dlemmermann
- JavaFX-based tutorials, William Antônio. Quick and easy tutorials from machine learning to IoT:
<https://fxapps.blogspot.com>
<https://github.com/jesuino>
Twitter:@William_Antonio
- JavaFX examples using the Ceylon language, Gavin King:
<https://github.com/ceylon/ceylon-examples-javafx>
Twitter:@1ovthafew
- Building the MyDevoxx App with JavaFX and Gluon, Johan Vos and Eugene Ryzhikov:
https://www.youtube.com/watch?v=1I38pmEnfVQ&list=PLy7t4z5SYNaRN2IDmjq_SSrb9Sph4zAC&index=12
Johan Vos: @johanvos
Eugene Ryzhikov: @eryzhikov
- The latest JavaFX Daily!, Michael Heinrichs. Newsfeed topics on none other than JavaFX!: <http://paper.li/net0pyr/1312275601#/>
Twitter: @net0pyr

Tools, Applications, and Libraries

- TilesFX, Gerrit Grunwald. A JavaFX library containing tiles for dashboards:
<https://bintray.com/hansolo/tilesfx/tilesfx/1.4.5>
Twitter:@hansolo_
- AnchorFX, Alessio Vinerbi. Docking framework for JavaFX platform:
<https://github.com/alexbodogit/AnchorFX>
Twitter:@Alessio_Vinerbi
- JFXGL, Jeff Martin. JFXGL is a cross-platform solution for integrating modern-style OpenGL rendering (v3.2+) with JavaFX UIs in JVM 8 applications. This project is intended for desktop use, and all three major platforms are supported: Linux, OS X, and Windows:
<https://bitbucket.org/cuchaz/jfxgl>
Twitter:@cuchaz
- OpenJFX 8 Nightly and Stable (8u60) builds, Chris Newland- Builds of OpenJFX 8 for Linux amd64 (Desktop), OS X, x86egl (Monocle), and armv6hf (Raspberry Pi):
<https://chriswhocodes.com>
Twitter:@chriswhocodes

- FontAwesomeFX, Jens Deter. A library with popular font packs used to apply to JavaFX buttons and labels:
<https://bitbucket.org/Jerady/fontawesomefx>
Twitter: @FontAwesomeFX
- FlexGanttFX and CalendarFX, Dirk Lemmermann. Professional quality custom controls for Gantt charts and planning calendars. Used in commercial applications:
<http://dlsc.com>
Twitter: @dlemmermann
- FXEventBus, Almas Baimagambetov: Simple event bus implementation based on JavaFX event dispatching mechanism.
GitHub: <https://github.com/AlmasB/FXEventBus>
Twitter: @AlmasBaim
- FXGL, Almas Baimagambetov: JavaFX game development framework:
GitHub: <https://github.com/AlmasB/FXGL>
Video: <https://www.youtube.com/almasb0/videos>
Twitter: @AlmasBaim
- JStackFX, Thierry Wasylczenko: A tool for analyzing thread dumps:
<https://github.com/twasy1/jstackfx>
Twitter: @twasy1
- SlideshowFX, Thierry Wasylczenko: Create interactive slide decks for presentations:
<http://slideshowfx.github.io>
Twitter: @SlideshowFX, @twasy1
- Orson Charts, David Gilbert: A 3D chart library for Java applications (JavaFX, Swing, or server-side):
<https://github.com/jfree/orson-charts>
- JavaFX Maven Docker Image, Jeff Miller:
<http://www.mymiller.name/wordpress/app/devops/javafx-maven-docker-image-openjfx/>
- UL Viewer, Yasumasa Suenaga. UL Viewer is log parser for JEP 158: Unified JVM Logging and JEP 271: Unified GC Logging:
GitHub: <https://github.com/YaSuenag/ulviewer/releases/tag/v0.1.0>
Twitter: @YaSuenag
- HeapStats, Yasumasa Suenaga. JVMTI agent and JavaFX analyzer to gather JVM runtime information for after-the-fact analysis:
GitHub: <https://github.com/HeapStats/heapstats>
Twitter: @YaSuenag
- JVx, SIB Visions. An MDI (multiple document interface) Library:
<https://blog.sibvisions.com/2015/06/22/jvx-javafxui-1-0/>
Twitter: @sibvisions
- JCSG-PathExtensions, Michael Hoffer. A JCSG extension library for working with simple paths (linearize SVG paths, extrude, and extend):
GitHub: <https://github.com/miho/JCSG-PathExtensions>
<https://bintray.com/miho/JCSG/JCSG-PathExtensions/0.1>
Twitter: @mihosoft

- VRL.Studio, Michael Hoffer. Innovative, intuitive, and powerful Visual IDE for rapid prototyping, learning, teaching, and experimentation:
<http://vrl-studio.mihosoft.eu>
Twitter: @mihosoft
- VWorkflows, Michael Hoffer. Library for interactive flow/graph visualization for building domain specific visual programming environments:
<https://bintray.com/miho/VWorkflows>
Twitter: @mihosoft
- Erlyberly, Andy Till. A debugger for erlang, elixir, and LFE using erlang tracing. Probably the easiest and quickest way to start debugging your nodes:
<https://github.com/andytill/erlyberly>
Twitter: @andy_till
- mapjfx 1.13.1 using OpenLayers 4.2.0, Peter-Josef Meisch. JavaFX implementation of a map using OpenLayers and JavaFX8:
<https://www.sothawo.com/projects/mapjfx/>
Twitter: @sothawo

Videos and Presentations on JavaFX

- TornadoFX, Edvin Syse. Lightweight JavaFX Framework for the Kotlin language:
GitHub: <https://github.com/edvin/tornadofx>
Twitter: @edvinsyse
Creating a LoginScreen in TornadoFX:
<https://www.youtube.com/watch?v=Jyuf4xn0oy4>
- DemoFX Part I, II, III, Chris Newland. Testbed for measuring JavaFX performance. JavaFX inspired by the demoscene:
<https://www.youtube.com/watch?v=N1rihYA8c2M>
<https://www.youtube.com/watch?v=WZfoj4GUFYM>
https://www.youtube.com/watch?v=9jztG_l8qrk
GitHub: <https://github.com/chriswhocodes/DemoFX>
Twitter: @chriswhocodes
- eteoBoard. Scrum Board Features, Saxonia Systems AG:
<https://www.youtube.com/watch?v=mX1SvXeUetQ>
<https://www.youtube.com/watch?v=4SEvfbudVwM>
- JavaFX in action, Alexander Casall:
<https://www.youtube.com/watch?v=WQPcuCEnipE>
- 3D Made Easy with JavaFX, Kevin Rushforth and Chien Yang:
https://www.youtube.com/watch?v=EBKhDv-_rIc
- Modeling, texturing, and lighting mesh geometry in JavaFX 3D, John Yoon:
<https://www.youtube.com/watch?v=lJvhHSC0dnCY>
- JavaFX 3D, F(x)yz Sampler preview, Jason Pollastrini:
https://www.youtube.com/watch?v=3z0fsV_8lGo
<https://www.youtube.com/watch?v=dEUZoclj2yo>

- Rigged hand animation with JavaFX and Leap Motion, José Pereda:
https://www.youtube.com/watch?v=8_xiv1pV3tI
- JavaFX Everywhere:
<https://www.youtube.com/watch?v=a3dAteWr40k>
- Integrating Spring Boot with JavaFX, MVP Java:
<https://www.youtube.com/watch?v=hjeSOxi3uPg>
<https://github.com/mvpjava>
- JavaFX Multiple Controllers, MVP Java:
<https://www.youtube.com/watch?v=osIRfgHTfyg>
- JavaFX, Switching Scenes Like A Boss!, MVP Java:
<https://www.youtube.com/watch?v=RifjriAxbw8>
- Converting a JavaFX app to Kotlin/TornadoFX one class at a time, Edvin Syse:
<https://www.youtube.com/watch?v=aSv5rwHxmxA>
- 3D floor visualization in the JavaFX PiDome client:
<https://www.youtube.com/watch?v=R0lXtpEFjqM>
<https://www.youtube.com/channel/UCcussRDCHBh0G7sIys9C7zg/feed>
- JavaFX material design: Setting up and making a login application, Genuine Coder:
<https://www.youtube.com/watch?v=1jiuM-gNyBc>
- JavaFX JFoenix Tutorial #9: Material Design Navigation Drawer:
<https://www.youtube.com/watch?v=tgV8dDP9DtM>
- iOS-Like Smooth Transitions with JavaFX, Bekwam, Inc.:
<https://www.youtube.com/watch?v=hpcpCGwPepM>
<https://www.youtube.com/channel/UCk2pGxJ397c5aC5-WhGPpog>
Twitter: @bekwaminc
- Let's Get Wet! AquaFX and Best Practices for Skinning JavaFX Controls, Claudine Zillmann and Hendrik Ebbers:
<https://www.youtube.com/watch?v=ucvdYGLWLTo>
Twitter: @etteClaudette, @hendrikEbbers
- Test-Driven Development with JavaFX, Sven Ruppert, and Hendrik Ebbers:
<https://www.youtube.com/watch?v=PaB0t8NP30c&t=1479s>
Twitter: @SvenRuppert, @hendrikEbbers
- The Moon, Jupiter, and Beyond: JavaFX in the Final Frontier, Sean Phillips:
<https://www.youtube.com/watch?v=WaybbwQNNOA>
- Sean Phillips: JavaFX In Action:
<https://www.youtube.com/watch?v=rfyHcsffQyE>
- JavaFX Deep Space Trajectory Explorer, Sean Phillips:
<https://www.youtube.com/watch?v=M0tQ1PC1xT8>
<https://www.youtube.com/channel/UCm8fN1bUnpIaz2-ycI5L45w>
- NASA Mission Software Development on the Eights: Java 8, JavaFX 8, and NetBeans 8:
<https://www.youtube.com/watch?v=609V5nc13h8>

- Dex, Patrick Martin: A data visualization tool written in Java/Groovy/JavaFX capable of powerful ETL and publishing web visualizations:
<https://dexvis.net>
https://www.youtube.com/watch?v=r54dsc58c_s
<https://github.com/PatMartin/Dex>
- TUT2097 Are You Listening? JavaFX Binding Techniques for Rich Client UIs, Paul and Gail Anderson:
<https://www.youtube.com/watch?v=lnvAKjZaRpE>
<http://asgteach.com>
Paul Anderson Twitter: @Paul_ASGTeach
Gail Anderson Twitter: @gail_asgteach

Index

A

Animation
JavaFX transition classes, 256
keyframes, 255
key values, 255
timeline, 256
transition (*see* Transition classes) (*see also* Compound transitions)
Application. *See* JavaFX, application lifecycle
Arduino
board, 391–393
Due, 392
IDE
Blink sketch, 400–401
execution, 398–400
MacOS X/Linux, 398
Windows, 396, 398
Uno, 391
Web Editor, 394–396
Arduino 101/Genuino 101, 392
AsciidocFX, 542
Audio playing
anchorPt variable, 302
close button, 313
drag-and-drop support, setting up, 308
file drag and drop, 308
media metadata, 308
MP3 player
application instance variables, 302
close button, 287
custom button panel creation, 305–306
play progress, rewind, and fast
forward, 307
progress and seek position slider
control, 287
source code, 287–296, 298–300
spectrum area chart visualization, 304
stage window, setting up, 303–304
stop, play, and pause buttons, 286

playMedia() method, 309
rewinding, 309
visualization updation, AudioSpectrumListener interface, 310–313
Audio spectrum data, 310–311

B

BasketStats app
adding model, 447–450
board view modification
board presenter, 458–462
ExpansionPanel controls, 457
game annotation view, 463
game events view, 463
game evolution view, 463
scene builder, 457
creation
AppViewManager class, 445–447
default build.gradle file, 444–445
Gluon plug-in, 441
MobileApplication class, 444
set name and location, 442
set views names, 443
deploy to mobile, 464
main view modification
main presenter, 454–455
scene builder, 453
storage service, 456
service adding, 450–452
Bloom uploader, 542
Body handlers, 340–341
BorderPane layout, 144–145
Bounding rectangle in parent, 87
Builder classes, 71, 84, 527
Building and running, project
Arduino3D class, 425, 426
OrientationFX class, 420, 426
sub-scene, 428
working process, 421–424

C

Capture variables, 106–107
 Cartesian coordinate system, 69
 Closed captioning video, 323–324
 Complex shapes
 cubic curve, 83
 donut, 86–87
 example, 79
 ice cream cone shape, 84–85
 smile, 86
 start() method, 80–83
 support, 79

Compound transitions, 261–262. *See also* Transition classes
 createClosedCaptionArea() method, 325
 createMediaView() method, 322
 Cubic curve, 83
 Cycle method, 94

D

Default methods
 cats example, 111–112
 code example, 112–115
 code explanation, 116–117
 Defender methods, 111
 Dependency injection, 164, 170
 disconnect() method, 413
 Domain-specific language, 532
 Donut shape, 86–87
 Drag and Drop, 249
 DragOver, 248
 TransferMode.LINK, 248
 DragBoard, 248
 Drawing lines
 common properties, 77
 java, 75–76
 lambda expressions, 77
 offset property, 77
 root node, 76
 setStrokeWidth() method, 77
 Drawing rectangles, 78
 Drawing shapes
 complex, 79–88
 rectangles, 78
 Drawing text
 apply effects, 100
 change text fonts, 97–100
 create random values, 96
 example, 95–96
 3D scene
 DrawMode, 383–387
 first person movement
 keyboard event handler, 376–377
 mouse event handler, 377–379

initialize with camera, 367–369
 MeshViews, 383–384, 386–387
 mouse type events, 375–376
 point light, 389–390
 roll camera, 388
 rotating, 473–475
 touching, 473–475
 TriangleMesh class, 380
 winding and wuthering, 380, 382–383
 zooming, 473–475
 DukeScript, 531

E

Event-driven architecture (EDA), 283

F

First person movement
 keyboard event handler, 376–377
 mouse event handler, 377–379
 FlowPane layout, 144
 Form-type application, 146–147
 column constraints, 149
 gridlines visible, 149
 output, 148
 top banner area, 150
 Fuzzy, 71–72
 FXDesktopSearch, 541
 FXForm2, 532

G

Gestures
 events, 468
 touch events
 animate circle, 469–471
 run, 471
 touch point, 468
 Getter method, 118
 GluonHQ company, 150
 Gluon Mobile, 531
 development tool, 438
 Glisten user interfaces, 439–440
 IDE plug-ins, 438
 license, 441
 Gradient color
 create, 92
 linear, 92
 radial gradient, 93
 reflective cycle, 94
 semitransparent, 94
 stop color, 92
 Gradle, 1
 -based project, 38
 installing, 22

GridPane layout
 column constraints, 145, 149
 form-type application, 146–147
 gridlines visible, 149
 output, 148
 top banner area, 150

Griffon, 531

Grouped primitives, 373–374

GUI frameworks, 531
 DukeScript, 531
 FXForm2, 532
 Gluon Mobile, 531
 Griffon, 531
 JavaFX MVP framework, 532
 jpro, 531
 NetBeansIDE-AfterburnerFX-Plugin, 532
 Open Dolphin, 531
 ReduxFX, 532
 TornadoFX, 531

H

Hands tracking project
 application class, 485–487
 3D model classes
 forearm, 485
 joint, 484–485
 phalanx, 484
 Utils, 483, 486–487

LeapListener class, 480–483
 Pair class, 482–483
 run, 487–488

HBox
 create borders, 138
 four rectangles example, 136, 138–140
 set spacing, 139
 shape nodes, 136
 single-argument constructor, 137
 total width in pixels, 139–140

Hello Mobile World
 build.gradle file, 433
 jfxmobile plug-in, 435
 main class, 434
 running androidInstall, 436
 signing configuration, 437

HTML5 analog clock, 345–347, 349

HTML DOM content
 body handlers, 340–341
 getDocument(), 331
 HttpClient.Builder class, 337
 HTTP GET request, 340
 HTTP POST request, 341–342
 HttpRequest API, 337–338
 HttpRequest.Builder request property
 methods, 339

JavaScript
 bridge, 332
 function, 333
 to java, 333–334
 jdkg.incubator.httpclient, 335–336
 JSON text, 335
 RESTful requests, 339
 String object, 332

I

Ice cream cone shape, 84–85
 ImageInfo.java, 252
 Images
 common parameters, 228
 display, 230
 getClassLoader() method, 229
 load, 228
 loading from a web server, 228–230, 264
 loading from the classpath, 228–229, 264
 loading from the file system, 228–229,
 231–232, 264
 resources directory, 229–230
 toURL() method, 229
 web server, 228–229
 ImageView. *See* Images
 ImageViewButtons.java, 250–252
 Inkscape and SVG, 349
 Integrated development environments
 (IDEs), 1–2, 526

J

JAR hell, 52
 Java 8
 books and magazines, 543
 JavaFX printing API, 527
 lambda expressions, 528
 Nashorn, 529
 Java 9
 API documentation, 525
 features, 525
 IDEs, 526
 Jigsaw, 526
 JavaBeans, 118, 119, 130
 java.beans.PropertyChangeSupport, 118. *See also*
 Properties
 Java development kit (JDK), 1
 Javadoc API documentation, 525
 JavaFX
 application lifecycle, 237
 applications, 541–542
 audio player, 286
 books and magazines, 543
 Cartesian *vs.* screen coordinate system, 69

■ INDEX

- JavaFX (*cont.*)
 Charts API, 414, 416, 418–419
 color, 527
 communities, 540
 3D, 537–538
 deploying applications, 526
 2D shapes, 527
 embedded, 539
 fuzzy, 71–72
 gaming, 538
 JavaBean, 118–121
 JavaFXPorts, 431–432
 JFXGL, 545
 layouts, 530
 line, 69
 media, 283, 535
 objects, 71
 operating system style guidelines, 535
 printing, 527
 properties and bindings, 529
 read-only properties, 119
 read/writable properties, 119
 transition classes, 256
 videos and presentations, 547–548
 WebView node, 536–537
 wrapper objects, 118
 2.x builder classes, 527
 yFiles, 534
javafx.animation.FadeTransition, 256
javafx.animation.PathTransition, 256
javafx.animation.ScaleTransition, 256
javafx.animation.Transition, 257, 261
javafx.animation.TranslateTransition, 256
javafx.beans.property.ReadOnlyBooleanWrapper, 118
javafx.beans.property.ReadOnlyDoubleWrapper, 118
javafx.beans.property.ReadOnlyIntegerWrapper, 118
javafx.beans.property.ReadOnlyStringWrapper, 119
javafx.beans.property.SimpleBooleanProperty, 118
javafx.beans.property.SimpleDoubleProperty, 118
javafx.beans.property.SimpleIntegerProperty, 118
javafx.beans.property.SimpleStringProperty, 109
JavaFX HelloWorld application, 30
 command-line prompt, 34–40
Netbeans IDE
 Java Platform Manager, 31
 launch, 30
 Main Project, 33
 newly created HelloWorld project, 31
 project and categories, 30
 Project Location and Project Folder fields, 30
 Project properties, 32
source code
 JavaFX node, 42–43
 main() method, 41
 overridden start() method, 41
 packaging, 43–44
javafx.scene.image.ImageView, 230, 236
Java IoT, 539
Java/JavaFX web-based APIs, 327
Java 8 JDK
 downloading required software, 1–2
 environment variables, 14–15
 MacOS X or Linux, 19–22
 Windows, setup, 16–19
 installation
 complete, 7
 destination folder, 6
 next button, 3
 optional features, 4
 progress bar, 5
 security warning, 3
Java 9 JDK, 3
 on Linux, 11
 Fedora, CentOS, Oracle Linux, 11
 Red Hat alternatives, 12
 Ubuntu/Debian, 13–14
 on MacOS X, 7, 9–11
 on Microsoft Windows, 3, 5–6
Java modules and class, 328–329
Java Specification Request (JSR) 103, 335
JFrog, 55
JideFX, 534
Jigsaw, 526
 automatic modules, 47
 benefits, 48–49
 definition, 48
 drawbacks, 49
 JAR hell, 52
 Java 9, 49
 jdeps tool, 50–51
 kill switch, 52
 Maven coordinates, 55
 OSGi, 53–54
 repository, 55
jpro, 531
JylooSoftware, 534

■ K

- Keyboard event handler, 376–377
Keyframes, 255
Key values, 255

■ L

- Lambda expressions, 77, 107, 117
 aggregate operations, 108–111
 anonymous inner class, 104
 binding
 bidirectional, 123
 high-level, 124
 low-level, 124

closures, 104

Greek symbol, 103

JSR 103, 335

parallel streams, 110–111

pipeline operations, 108

project, 528

syntax, 104–105

Layouts

BorderPane, 144–145

FlowPane, 144

GridPane, 145, 147–150

HBox, 136–140

VBox, 140–143

Lazy evaluation, 107

Leap Motion controller

Cartesian coordinate system, 478

device and box, 476

diagnostic visualizer application, 477

hands tracking project (*see* Hands tracking project)

LEDs, 477

SDK

download, 478

JAR, 478

JavaFX threads, 479

onConnect() method, 479

Lifecycle

init(), 238

start(), 235–239, 242

stop(), 237–238

Logon dialogs

class member variables, 131

code, 126–129, 131

display, 126

start() method, 131–134

M

Madgwick library, 402

Maven, 47, 54, 55, 68

Media events, 283–284

Media marker events, 324

media player, 283, 285, 287, 293, 301, 302, 305–308, 310, 313, 314

Status.PLAYING, 292, 293, 305, 306

MediaPlayer events, 284

MediaView node, 321

Method reference, 106

Modellus application, 542

Modular application

application code, 64–65

compile code, 65

copy resources, 65

JAR

package application, 66–67

run application, 67

module definition, 63–64

project structure, 63

run application, 66

Module path, 56

Module project

location, 407

module-info class, 409

naming, 408

NetBeans, 407

OrientationFX class, 409

Modules

application, 59

automatic, 59, 62

definition, 57

exports, 58

platform, 59–61

requires, 57

unnamed, 59, 62

Mouse event, 375–376

Mouse event handler, 377–379

MPEG-4, 314

Myst game, 257

N

Nano Editor's keyboard commands, 20

Nashorn, 529

NetBeans IDE, 1, 45

installation

check for updates option, 26

destination directory, 25

license agreement, 24

NetBeansIDE-AfterburnerFX-Plugin, 532

next button, 23

progress bar, 27

setup complete, 28

start page, 29

Nonlocal variables. *See* Capture variables

O

Operating system style guidelines, 535

Orientation visualizer, 401–402, 404–405

OSGi, 53–54

P, Q

Painting colors, 88–91

Parallelism, 110

PhotoViewer2

loadAndDisplayImage() method, 262

transitionByFading() method, 263

Photo Viewer application

adjust colors, 233

ImageInfo.java, 235, 252

ImageViewButtons.java, 235, 250–252

■ INDEX

- Photo Viewer application (*cont.*)
 loads images, 231
 photo-viewer.css, 235, 253–254
 PhotoViewer.java, 235
 Color Adjust menu, 243–244
 constants, 235–236
 createSliderMenuItem() method, 244
 factory function, 238
 file menu, 239
 isValidImageFile() method, 250
 loadAndDisplayImage()
 method, 247–248
 main(), 237–238
 menu options, 240–241
 progress indicator, 239
 rotateImageView() method, 243
 rotate menu, 242
 save image, 241
 setupDragNDrop() method, 248, 250
 start(), 237–238
 state variables, 235–236
 stop(), 237–238
 updateSliders() method, 245
 URL to load and render image, 248
 wireupRotateAngleBy(), 242
 wireupUIBehavior() method, 247
previous/next image buttons, 233
progress indicator, 232
resize image, 232
rotate image, 232
Save As option, 233
UML class, 233–234
- PiDome, 539, 541
Plain old Java object (POJO), 235
playMedia() method, 309–310
Point-and-click game
 fade transition, 259, 261
 JavaFX Transition APIs, 257
 scale transition, 258, 261
 SVG nodes, 257–258, 260–261
- Point light, 389–390
- Primitives
 Cylinder object, 369–370
 grouped, 373–374
 transforming cube and sphere, 372–373
 translate and rotate, 371–372
- Printing
 APIs, 268
 MyPrinterHelper.createPages(), 267
 Printer and PrinterJob
 configuration, 272–273
 PrinterAttributes, 269–271
 web page, 274
- Properties, 103–134
Property change support
 change() method, 121–122
 invalidated () method, 122
- R
- Radial gradient, 93
Range offset, 92
Reflective cycle gradients, 94
RESTful requests, 339
Retained-mode approach, 99
Rich Internet Application (RIA), 227, 283, 327
Robo4J, 539
Roll camera, 388
Rotate gesture, 473–474
- S
- Scene Builder
 BorderPane, 153–155
 Controller class, 152
 dependency injection, 164
 download, 151
 first name label, 160
 fx:id name, 165–166
 @FXML annotation, 170
 FXMLContactForm.java, 168–169
 FXML file, 162, 164
 FXMLLoader.load() method, 170
 GridPane layout, 156, 158–159
 Hierarchy panel, 152
 install, 151
 last name label, 162
 launch, 151
 subsections, 151
 user interface, 152–153, 162
 WYSIWYG editors, 150
- Scene.lookup() method, 304
Screen coordinate system, 70
Semitransparent gradients, 94
SequentialTransition, 261–264. *See also* Transition classes
Serial communications, 409, 411–413
Serial reading, 406
Setter methods, 73, 118
Single abstract method (SAM), 107
Smile shape, 86
Software installation
 Java 8 JDK (*see* Java 8 JDK)
 Java 9 JDK (*see* Java 9 JDK)
 NetBeans IDE (*see* NetBeans IDE)
 operating systems, 2

start() method, 363–364
 ATTEMPTS property, 134
 CSS file, 132
 password field, 133
 rectangular background, 132
 SVG path notation, 133
 User class, 131
 Stop color, 92
 streamToString() method, 353
 SyntheticaFX, 534

■ T

Testing Serial Comms, 414
 Text fonts, change, 97–100
 Timeline, 256
 TornadoFX, 531
 Touch events
 animate circle, 469–471
 run, 471
 Touch gesture, 473
 Touch point, 468
 Transition classes, 227, 256, 258–261
 TriangleMesh class, 380

■ U

Ubuntu/Debian system, 13–14
 UML class, 233–234
 User interface (UI)
 customize, 532–534
 patterns, 117

■ V

VBox layout
 child node, 140
 five-pixel spacing, 142

four rectangles example, 140–142
 total width in pixels, 143
 Video player
 closed captioning video, 323–324
 MediaView node, 321–322
 MPEG-4, 314
 source code, 316–319
 stage window, full-screen mode, 320
 start() method, 320
 VP6 .flv, 314
 vi Editor's keyboard commands, 21
 Virtual extension methods, 111
 VP6 .flv, 314

■ W, X, Y

Weather Widget, 351–362
 WebDocPrinter
 application, 275, 277–280
 print modes, 275
 WebView, 281
 WebEngine API, 330–331
 WebEvents, 350, 351
 WebSockets
 client-side sockets, 343, 345
 HTML5 content, 345
 server-side sockets, 342–343
 Winding
 addAll() method, 382
 buildPyramid(), 382
 getFaces().addAll(), 382
 square pyramid, 380
 TriangleMesh object, 380–382
 WYSIWYG editors, 150

■ Z

Zoom gesture, 475