

ORACLE®

"Provides a clear and concise introduction to JavaFX and will help a developer to quickly get up to speed with JavaFX. The author makes learning a new and exciting technology easy with practical examples. Take your skills to the next level—buy this book and learn JavaFX."

—Bobby Higbea, Software Developer, Tampa, FL

Covers Release 1.3

# JavaFX A Beginner's Guide



Build Interactive Desktop, Browser, and Mobile Applications

J. F. DiMarzio

ORIGINAL • AUTHENTIC  
**Oracle Press**  
ONLY FROM McGRAW-HILL

**ORACLE®**

*Oracle Press™*

# JavaFX™

## A Beginner's Guide

## About the Author

**J. F. DiMarzio** is the author of eight development and architecture titles. Born in Boston, Massachusetts, he moved to Central Florida in the mid-1990s to work in the area's emerging technology market. Now a leading web and mobile development resource, DiMarzio works for a Fortune 500 company as a senior e-commerce developer. His previous titles, including *The Debugger's Handbook* and *Android: A Programmer's Guide*, have been sold worldwide, used as textbooks, and translated into multiple languages.

## About the Technical Editor

**Joshua Flood** has spent more than a decade professionally developing and supporting dynamic web applications using many technologies, including JavaFX, Java Servlets, JavaScript, PHP, Perl, and Flex. Joshua has extensive experience in all aspects of web development—from small standalone site development through large-scale dynamic web content delivery systems. In addition, he has helped architect scalable high-availability sites that handle traffic for clients around the world.



*Oracle Press*<sup>TM</sup>

# JavaFX™

## A Beginner's Guide

*J. F. DiMarzio*



New York Chicago San Francisco  
Lisbon London Madrid Mexico City  
Milan New Delhi San Juan  
Seoul Singapore Sydney Toronto

Copyright © 2011 by The McGraw-Hill Companies, Inc. (Publisher). All rights reserved. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

ISBN: 978-0-07-174240-5

MHID: 0-07-174240-9

The material in this eBook also appears in the print version of this title: ISBN: 978-0-07-174241-2,  
MHID: 0-07-174241-7.

All trademarks are trademarks of their respective owners. Rather than put a trademark symbol after every occurrence of a trademarked name, we use names in an editorial fashion only, and to the benefit of the trademark owner, with no intention of infringement of the trademark. Where such designations appear in this book, they have been printed with initial caps.

McGraw-Hill eBooks are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. To contact a representative please e-mail us at [bulksales@mcgraw-hill.com](mailto:bulksales@mcgraw-hill.com).

Information has been obtained by Publisher from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, Publisher, or others, Publisher does not guarantee to the accuracy, adequacy, or completeness of any information included in this work and is not responsible for any errors or omissions or the results obtained from the use of such information.

Oracle Corporation does not make any representations or warranties as to the accuracy, adequacy, or completeness of any information contained in this Work, and is not responsible for any errors or omissions.

#### TERMS OF USE

This is a copyrighted work and The McGraw-Hill Companies, Inc. (“McGrawHill”) and its licensors reserve all rights in and to the work. Use of this work is subject to these terms. Except as permitted under the Copyright Act of 1976 and the right to store and retrieve one copy of the work, you may not decompile, disassemble, reverse engineer, reproduce, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish or sublicense the work or any part of it without McGraw-Hill’s prior consent. You may use the work for your own noncommercial and personal use; any other use of the work is strictly prohibited. Your right to use the work may be terminated if you fail to comply with these terms.

THE WORK IS PROVIDED “AS IS.” McGRAW-HILL AND ITS LICENSORS MAKE NO GUARANTEES OR WARRANTIES AS TO THE ACCURACY, ADEQUACY OR COMPLETENESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. McGraw-Hill and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither McGraw-Hill nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. McGraw-Hill has no responsibility for the content of any information accessed through the work. Under no circumstances shall McGraw-Hill and/or its licensors be liable for any indirect, incidental, special, punitive, consequential or similar damages that result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.

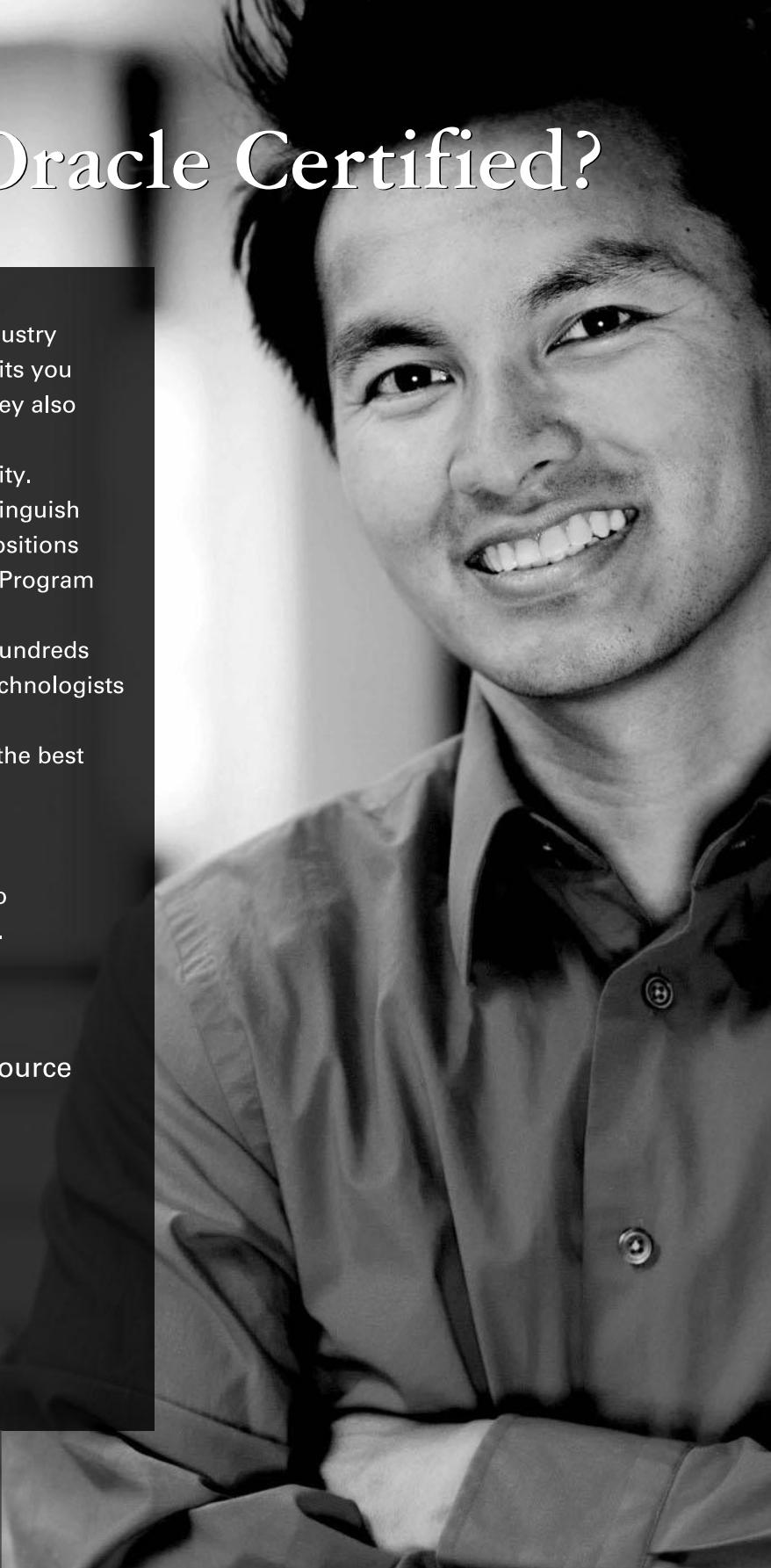
# Are You Oracle Certified?

Professional development and industry recognition are not the only benefits you gain from Oracle certifications. They also facilitate career growth, improve productivity, and enhance credibility. Hiring managers who want to distinguish among candidates for critical IT positions know that the Oracle Certification Program is one of the most highly valued benchmarks in the marketplace. Hundreds of thousands of Oracle certified technologists testify to the importance of this industry-recognized credential as the best way to get ahead—and stay there.

For details about the Oracle Certification Program, go to [oracle.com/education/certification](http://oracle.com/education/certification).

Oracle University —  
Learn technology from the source

**ORACLE®**  
UNIVERSITY

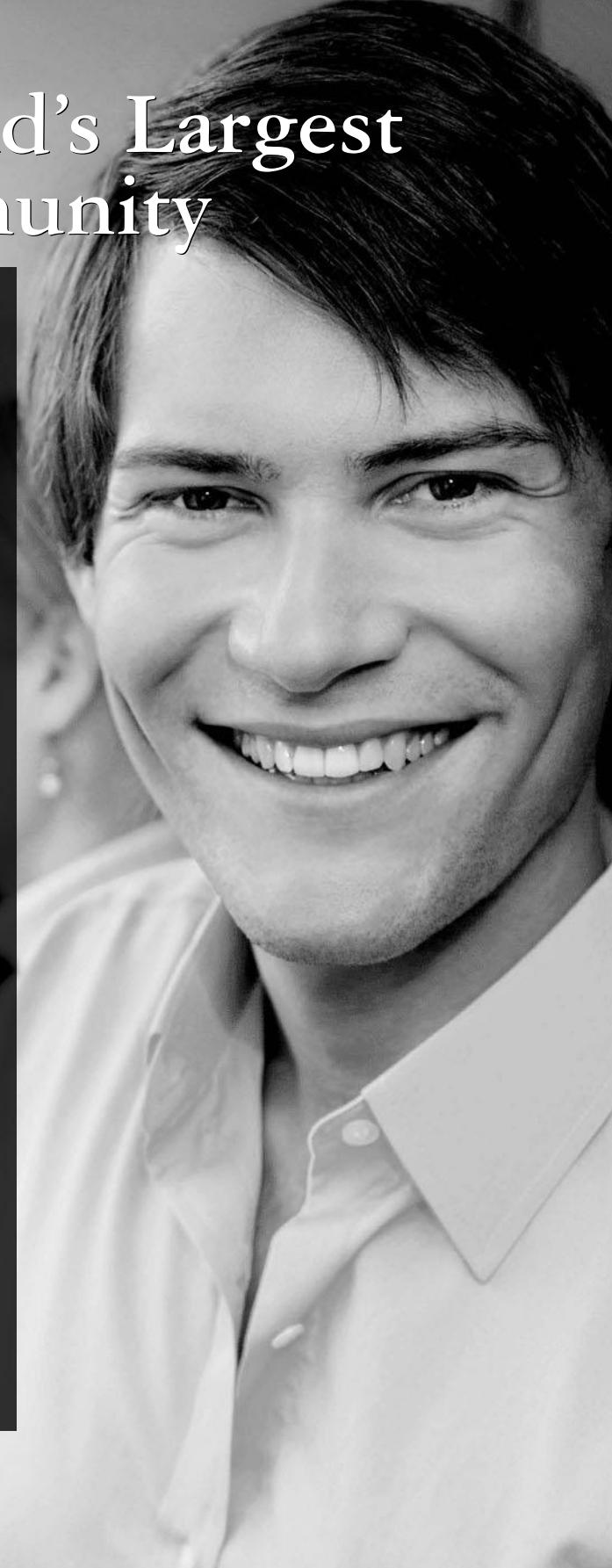


# Join the World's Largest Oracle Community

With more than 5 million members, Oracle Technology Network ([otn.oracle.com](http://otn.oracle.com)) is the best place online for developers, DBAs, and architects to interact, exchange advice, and get software downloads, documentation, and technical tips — all for free!

Registration is easy;  
join Oracle Technology Network today:  
[otn.oracle.com/join](http://otn.oracle.com/join)

**ORACLE®**  
TECHNOLOGY NETWORK

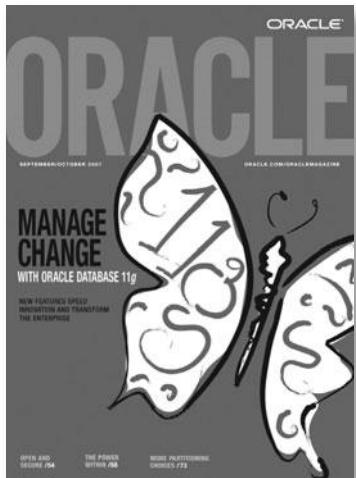


# GET YOUR FREE SUBSCRIPTION TO *ORACLE MAGAZINE*

***Oracle Magazine*** is essential gear for today's information technology professionals.

Stay informed and increase your productivity with every issue of *Oracle Magazine*.

Inside each free bimonthly issue you'll get:



- Up-to-date information on Oracle Database, Oracle Application Server, Web development, enterprise grid computing, database technology, and business trends
- Third-party news and announcements
- Technical articles on Oracle and partner products, technologies, and operating environments
- Development and administration tips
- Real-world customer stories

If there are other Oracle users at your location who would like to receive their own subscription to *Oracle Magazine*, please photocopy this form and pass it along.

**ORACLE**  
MAGAZINE

## Three easy ways to subscribe:

### ① Web

Visit our Web site at [oracle.com/oraclemagazine](http://oracle.com/oraclemagazine)  
You'll find a subscription form there, plus much more

### ② Fax

Complete the questionnaire on the back of this card  
and fax the questionnaire side only to **+1.847.763.9638**

### ③ Mail

Complete the questionnaire on the back of this card  
and mail it to **P.O. Box 1263, Skokie, IL 60076-8263**

**ORACLE®**

# Want your own FREE subscription?

To receive a free subscription to *Oracle Magazine*, you must fill out the entire card, sign it, and date it (incomplete cards cannot be processed or acknowledged). You can also fax your application to +1.847.763.9638. Or subscribe at our Web site at [oracle.com/oraclemagazine](http://oracle.com/oraclemagazine)

Yes, please send me a FREE subscription *Oracle Magazine*.

No.

From time to time, Oracle Publishing allows our partners exclusive access to our e-mail addresses for special promotions and announcements. To be included in this program, please check this circle. If you do not wish to be included, you will only receive notices about your subscription via e-mail.

Oracle Publishing allows sharing of our postal mailing list with selected third parties. If you prefer your mailing address not to be included in this program, please check this circle.

If at any time you would like to be removed from either mailing list, please contact Customer Service at +1.847.763.9638 or send an e-mail to [oracle@halldata.com](mailto:oracle@halldata.com). If you opt in to the sharing of information, Oracle may also provide you with e-mail related to Oracle products, services, and events. If you want to completely unsubscribe from any e-mail communication from Oracle, please send an e-mail to: [unsubscribe@oracle-mail.com](mailto:unsubscribe@oracle-mail.com) with the following in the subject line: REMOVE [your e-mail address]. For complete information on Oracle Publishing's privacy practices, please visit [oracle.com/html/privacy/html](http://oracle.com/html/privacy/html)

Would you like to receive your free subscription in digital format instead of print if it becomes available?  Yes  No

## YOU MUST ANSWER ALL 10 QUESTIONS BELOW.

**①** WHAT IS THE PRIMARY BUSINESS ACTIVITY OF YOUR FIRM AT THIS LOCATION? (check one only)

- 01 Aerospace and Defense Manufacturing
- 02 Application Service Provider
- 03 Automotive Manufacturing
- 04 Chemicals
- 05 Media and Entertainment
- 06 Construction/Engineering
- 07 Consumer Sector/Consumer Packaged Goods
- 08 Education
- 09 Financial Services/Insurance
- 10 Health Care
- 11 High Technology Manufacturing, OEM
- 12 Industrial Manufacturing
- 13 Independent Software Vendor
- 14 Life Sciences (biotech, pharmaceuticals)
- 15 Natural Resources
- 16 Oil and Gas
- 17 Professional Services
- 18 Public Sector (government)
- 19 Research
- 20 Retail/Wholesale/Distribution
- 21 Systems Integrator, VAR/VAD
- 22 Telecommunications
- 23 Travel and Transportation
- 24 Utilities (electric, gas, sanitation, water)
- 98 Other Business and Services \_\_\_\_\_

**②** WHICH OF THE FOLLOWING BEST DESCRIBES YOUR PRIMARY JOB FUNCTION? (check one only)

- CORPORATE MANAGEMENT/STAFF
- 01 Executive Management (President, Chair, CEO, CFO, Owner, Partner, Principal)
  - 02 Finance/Administrative Management (VP/Director/Manager/Controller, Purchasing, Administration)
  - 03 Sales/Marketing Management (VP/Director/Manager)
  - 04 Computer Systems/Operations Management (CIO/VP/Director/Manager MIS/IT, Ops)
- IS/IT STAFF
- 05 Application Development/Programming Management
  - 06 Application Development/Programming Staff
  - 07 Consulting
  - 08 DBA/Systems Administrator
  - 09 Education/Training
  - 10 Technical Support Director/Manager
  - 11 Other Technical Management/Staff
  - 98 Other

**③** WHAT IS YOUR CURRENT PRIMARY OPERATING PLATFORM (check all that apply)

- 01 Digital Equipment Corp UNIX/VAX/VMS
- 02 HP UNIX
- 03 IBM AIX
- 04 IBM UNIX
- 05 Linux (Red Hat)
- 06 Linux (SUSE)
- 07 Linux (Oracle Enterprise)
- 08 Linux (other)
- 09 Macintosh
- 10 MVS
- 11 Netware
- 12 Network Computing
- 13 SCO UNIX
- 14 Sun Solaris/SunOS
- 15 Windows
- 16 Other UNIX
- 98 Other
- 99 None of the Above

**④** DO YOU EVALUATE, SPECIFY, RECOMMEND, OR AUTHORIZE THE PURCHASE OF ANY OF THE FOLLOWING? (check all that apply)

- 01 Hardware
- 02 Business Applications (ERP, CRM, etc.)
- 03 Application Development Tools
- 04 Database Products
- 05 Internet or Intranet Products
- 06 Other Software
- 07 Middleware Products
- 99 None of the Above

**⑤** IN YOUR JOB, DO YOU USE OR PLAN TO PURCHASE ANY OF THE FOLLOWING PRODUCTS? (check all that apply)

- SOFTWARE
- 01 CAD/CAE/CAM
  - 02 Collaboration Software
  - 03 Communications
  - 04 Database Management
  - 05 File Management
  - 06 Finance
  - 07 Java
  - 08 Multimedia Authoring
  - 09 Networking
  - 10 Programming
  - 11 Project Management
  - 12 Scientific and Engineering
  - 13 Systems Management
  - 14 Workflow
- HARDWARE
- 05 Macintosh
  - 16 Mainframe
  - 17 Massively Parallel Processing

18 Minicomputer

- 19 Intel x86(32)
- 20 Intel x86(64)
- 21 Network Computer
- 22 Symmetric Multiprocessing
- 23 Workstation Services

SERVICES

- 24 Consulting
- 25 Education/Training
- 26 Maintenance
- 27 Online Database
- 28 Support
- 29 Technology-Based Training
- 30 Other
- 99 None of the Above

- 05 Hibernate
- 06 J++/J#
- 07 Java
- 08 JSP
- 09 .NET
- 10 Perl
- 11 PHP
- 12 PL/SQL
- 17 SQL
- 18 Visual Basic
- 98 Other

**⑥** WHAT IS YOUR COMPANY'S SIZE? (check one only)

- 01 More than 25,000 Employees
- 02 10,000 to 25,000 Employees
- 03 5,001 to 10,000 Employees
- 04 1,001 to 5,000 Employees
- 05 101 to 1,000 Employees
- 06 Fewer than 100 Employees

**⑦** DURING THE NEXT 12 MONTHS, HOW MUCH DO YOU ANTICIPATE YOUR ORGANIZATION WILL SPEND ON COMPUTER HARDWARE, SOFTWARE, PERIPHERALS, AND SERVICES FOR YOUR LOCATION? (check one only)

- 01 Less than \$10,000
- 02 \$10,000 to \$49,999
- 03 \$50,000 to \$99,999
- 04 \$100,000 to \$499,999
- 05 \$500,000 to \$999,999
- 06 \$1,000,000 and Over

**⑧** WHAT IS YOUR COMPANY'S YEARLY SALES REVENUE? (check one only)

- 01 \$500,000,000 and above
- 02 \$100,000,000 to \$500,000,000
- 03 \$50,000,000 to \$100,000,000
- 04 \$5,000,000 to \$50,000,000
- 05 \$1,000,000 to \$5,000,000

**⑨** WHAT LANGUAGES AND FRAMEWORKS DO YOU USE? (check all that apply)

- 01 Ajax
- 02 C
- 03 C++
- 04 C#
- 05 Python
- 06 Ruby/Rails
- 07 Spring
- 08 PHP
- 09 Struts

**⑩** WHAT ORACLE PRODUCTS ARE IN USE AT YOUR SITE? (check all that apply)

ORACLE DATABASE

- 01 Oracle Database 11g
- 02 Oracle Database 10g
- 03 Oracle9i Database
- 04 Oracle Embedded Database (Oracle Lite, Times Ten, Berkeley DB)
- 05 Other Oracle Database Release

ORACLE FUSION MIDDLEWARE

- 06 Oracle Application Server
- 07 Oracle Portal
- 08 Oracle Enterprise Manager
- 09 Oracle BPEL Process Manager
- 10 Oracle Identity Management
- 11 Oracle SOA Suite
- 12 Oracle Data Hubs

ORACLE DEVELOPMENT TOOLS

- 13 Oracle JDeveloper
- 14 Oracle Forms
- 15 Oracle Reports
- 16 Oracle Designer
- 17 Oracle Discoverer
- 18 Oracle BI Beans
- 19 Oracle Warehouse Builder
- 20 Oracle WebCenter
- 21 Oracle Application Express

ORACLE APPLICATIONS

- 22 Oracle E-Business Suite
- 23 PeopleSoft Enterprise
- 24 JD Edwards EnterpriseOne
- 25 JD Edwards World
- 26 Oracle Fusion
- 27 Hyperion
- 28 Siebel CRM

ORACLE SERVICES

- 29 Oracle Technology On Demand
- 30 Siebel CRM On Demand
- 31 Oracle Consulting
- 32 Oracle Education
- 33 Oracle Support
- 98 Other
- 99 None of the Above

*This book is dedicated to Suzannah, Christian, Sophia, and Giovanni.*

*This page intentionally left blank*

# Contents at a Glance

<b>1</b>	<b>Introduction to JavaFX</b>	<b>1</b>
<b>2</b>	<b>Setting the Scene</b>	<b>9</b>
<b>3</b>	<b>Hello World</b>	<b>27</b>
<b>4</b>	<b>Creating Shapes</b>	<b>51</b>
<b>5</b>	<b>Using Colors and Gradients</b>	<b>71</b>
<b>6</b>	<b>Using Images</b>	<b>85</b>
<b>7</b>	<b>Applying Effects and Transformations</b>	<b>101</b>
<b>8</b>	<b>Basic Animation</b>	<b>131</b>
<b>9</b>	<b>Using Events</b>	<b>147</b>
<b>10</b>	<b>Give It Some Swing</b>	<b>163</b>
<b>11</b>	<b>Custom Nodes and Overriding</b>	<b>183</b>
<b>12</b>	<b>Embedded Video and Music</b>	<b>201</b>
<b>13</b>	<b>Using JavaFX Layouts</b>	<b>219</b>
<b>14</b>	<b>Style Your JavaFX with CSS</b>	<b>233</b>

<b>A Deploying JavaFX</b>	.....	247
<b>B Node Property Reference</b>	.....	253
<b>C JavaFX Command-Line Arguments</b>	.....	267
<b>D Answers to Self Tests</b>	.....	279
<b>Index</b>	.....	291

# Contents

ACKNOWLEDGMENTS .....	xiii
INTRODUCTION .....	xv
<b>1 Introduction to JavaFX .....</b>	<b>1</b>
What Is JavaFX? .....	2
What Is Needed for JavaFX Development?	2
Required Skills and Knowledge .....	3
Required Software .....	4
Downloading and Installing the Required Software .....	4
NetBeans .....	4
Try This: Configure Your NetBeans .....	7
Chapter 1 Self Test .....	8
<b>2 Setting the Scene .....</b>	<b>9</b>
Creating a New JavaFX Project .....	10
The Empty JavaFX Project .....	12
Adding Working Files to Your Project .....	15
Exploring the Empty Project in NetBeans .....	15
Working with the Script File .....	19
The Comments .....	19
The package Statement .....	20
Your First Stage .....	21
Inserting the Stage Snippet .....	21
A JavaFX Script Primer .....	23
Name-Value Pairs .....	23
Compiling Your JavaFX Script .....	24
Chapter 2 Self Test .....	25

<b>3 Hello World .....</b>	<b>27</b>
Writing to the Screen .....	28
Adding Some Descriptive Comments .....	30
Adding the Stage and Scene .....	30
Adding Some Text .....	31
Try This: Create a TV Run Configuration .....	40
Adding a Function .....	40
Using bind with a Text Node .....	46
Chapter 3 Self Test .....	50
<b>4 Creating Shapes .....</b>	<b>51</b>
Drawing Shapes .....	52
Before You Begin .....	52
Lines and Polylines .....	53
Rectangles .....	61
Polygons .....	64
Arcs .....	65
Circles and Ellipses .....	68
Try This: Create Multiple Shapes .....	70
Chapter 4 Self Test .....	70
<b>5 Using Colors and Gradients .....</b>	<b>71</b>
Using Color .....	72
Predefined Colors .....	72
Mixing Colors .....	74
Using Gradients .....	77
LinearGradients .....	77
RadialGradients .....	81
Try This: Create a Custom Gradient .....	82
Chapter 5 Self Test .....	83
<b>6 Using Images .....</b>	<b>85</b>
The ImageView Node .....	86
The Image Class .....	87
JavaFX Production Suite .....	91
Using an FXZ File in JavaFX .....	96
Try This: Working with Different Image Types .....	99
Chapter 6 Self Test .....	99
<b>7 Applying Effects and Transformations .....</b>	<b>101</b>
Effects .....	106
Bloom .....	106
ColorAdjust .....	109
GaussianBlur .....	110
Glow .....	113

DropShadow .....	116
InvertMask .....	119
Lighting .....	120
SepiaTone .....	123
Transformations .....	125
XY Transformations .....	125
Rotation .....	127
PerspectiveTransform .....	128
Try This: Combining Multiple Effects .....	129
Chapter 7 Self Test .....	130
<b>8 Basic Animation .....</b>	<b>131</b>
Timelines .....	133
Animating Along a Path .....	139
Try This: Create a Path Animation .....	145
Chapter 8 Self Test .....	145
<b>9 Using Events .....</b>	<b>147</b>
What Are Events? .....	148
Mouse Events .....	148
Key Events .....	157
Chapter 9 Self Test .....	161
<b>10 Give It Some Swing .....</b>	<b>163</b>
What Is Swing? .....	164
Swing Components .....	165
SwingButton .....	166
SwingCheckBox .....	173
SwingComboBox and SwingComboBoxItem .....	176
Try This: Create an Application with Swing .....	180
Chapter 10 Self Test .....	181
<b>11 Custom Nodes and Overriding .....</b>	<b>183</b>
Overriding a Node .....	184
Creating a RoundButton .....	186
Creating a Custom Node .....	192
Try This: Create Your Own Shapes .....	199
Chapter 11 Self Test .....	199
<b>12 Embedded Video and Music .....</b>	<b>201</b>
Playing Video .....	203
Creating a Play/Pause Button .....	207
Creating a Progress Indicator .....	211
Playing Audio .....	216
Chapter 12 Self Test .....	218

<b>13 Using JavaFX Layouts .....</b>	<b>219</b>
The HBox .....	221
The VBox .....	225
Nested Layouts .....	227
Try This: Using Other Layouts .....	230
Chapter 13 Self Test .....	231
<b>14 Style Your JavaFX with CSS .....</b>	<b>233</b>
Adding a Style Sheet to Your Packages .....	235
Creating a Style .....	238
Using Your Styles .....	239
Creating Independent Style Classes .....	241
Try This: Experimenting with Styles .....	244
Chapter 14 Self Test .....	244
<b>A Deploying JavaFX .....</b>	<b>247</b>
Deploying JavaFX .....	248
<b>B Node Property Reference .....</b>	<b>253</b>
Node Properties .....	254
Mouse Events .....	256
Key Codes .....	258
MediaPlayer Properties .....	265
<b>C JavaFX Command-Line Arguments .....</b>	<b>267</b>
Command-Line Environment .....	268
javafxc .....	269
javafx .....	274
<b>D Answers to Self Tests .....</b>	<b>279</b>
Chapter 1 .....	280
Chapter 2 .....	280
Chapter 3 .....	281
Chapter 4 .....	282
Chapter 5 .....	283
Chapter 6 .....	283
Chapter 7 .....	284
Chapter 8 .....	285
Chapter 9 .....	286
Chapter 10 .....	286
Chapter 11 .....	287
Chapter 12 .....	288
Chapter 13 .....	289
Chapter 14 .....	290
<b>Index .....</b>	<b>291</b>



# Acknowledgments

I would like to thank everyone who participated in the creation of this book. My agent Neil Salkind; Joya, Megg, and the crew at McGraw-Hill; Josh Flood; Bart Reed; Tania Andrabi at Glyph International; and everyone at Studio B.

I would also like to thank my family—Suzannah, Christian, Sophia, and Giovanni—my co-workers Jeanwill, Jeff, Tyrone, Larry, Steve, Rodney, Kelly, Soma, Eric, Orlando, Michelle, Matt, Nishad, Sarah, as well as all my colleagues in Central Florida and anyone else I may have forgotten.

*This page intentionally left blank*



# Introduction

Welcome to *JavaFX: A Beginner's Guide*. This book has been designed to give you the best first step into the exciting new frontier of JavaFX development. JavaFX is a rich environment tool, and learning JavaFX is a must for anyone who wants to create immersive, interactive environments for users of any background.

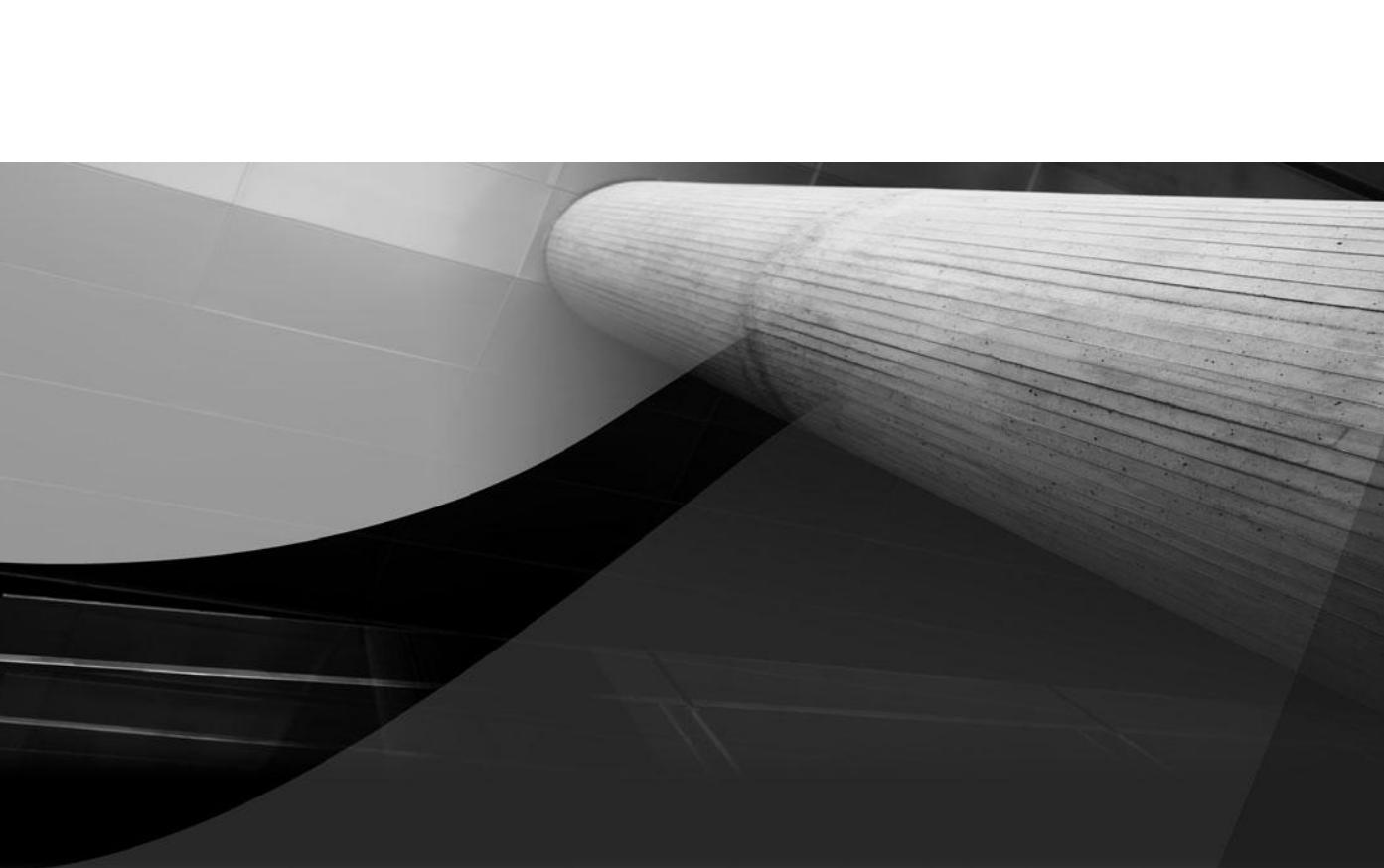
This book takes you through JavaFX in a logical manner. It begins by explaining the technology behind JavaFX. You will quickly move into installing the JavaFX development environment and tools. Although multiple development environments are available for JavaFX, the focus of this book is on teaching you the basics of NetBeans. NetBeans offers a rich, full-featured product that is easy to learn and will get you up and running in JavaFX in no time.

Most chapters also include a “Try This” section to help you practice what you have learned. The “Try This” sections are structured like a textbook in that you will be presented with tasks to complete on your own. In addition, each chapter has a “Self Test” section that provides ten quiz-style questions to further enhance your learning experience. Taking full advantage of the chapter questions and “Try This” exercises will give you a chance to refine your newly acquired skills and create your own applications.

Although this book is not an advanced programmer’s reference, you should possess certain skills to get the most from *JavaFX: A Beginner's Guide*. Foremost of these is Java programming fundamentals. Knowledge of Java classes and basic types will help you understand some of the concepts in this book more easily. Although JavaFX environments are written primarily in JavaFX Script, you can enhance the functionality of these environments using Java.

Any comments, questions, or suggestions about any of the material in this book can be sent directly to the author at [jfdimarzio@jfdimarzio.com](mailto:jfdimarzio@jfdimarzio.com).

*This page intentionally left blank*



# Chapter 1

## Introduction to JavaFX

## Key Skills & Concepts

- Installing JavaFX
- Installing NetBeans
- Using NetBeans

Welcome to *JavaFX: A Beginner's Guide*. I am sure you are anxious to begin your journey into the exciting world of JavaFX development, and this is the perfect place to start. Before you begin you need to have a fully capable development environment. This chapter will cover the basic knowledge needed to create and establish a JavaFX development environment that will allow you to create excitingly rich interactive applications. It will also answer many of the questions you may have about what JavaFX does, and how it does it.

## What Is JavaFX?

If you have ever used an Adobe Flash game, or seen an application in Microsoft Silverlight, then you have a pretty good idea of what JavaFX is. I know, it's not fair to compare JavaFX to these other environments, but if you have never seen JavaFX before, then I have to compare it to *something*.

JavaFX delivers full-featured, interactive experiences to users in much the same way as Flash or Silverlight. However, one of the major differences between JavaFX and the others is that JavaFX is platform independent. Because JavaFX is fully integrated with the Java Runtime, any device or system that can run Java can also serve up JavaFX experiences.

### NOTE

The JavaFX development environment currently allows for deployment on the Desktop, Web, television, and mobile devices.

## What Is Needed for JavaFX Development?

Before you jump right into development, you should examine the list of requirements as outlined in the following section. Think of them as the prerequisites for a successful and

rewarding learning process. You should have at least a basic knowledge of the following skills as well as access to the listed software.

## Required Skills and Knowledge

Prior development experience is not required to follow along with this book. If you have never created a single application or developed a basic web page, you will still have the skills needed to learn JavaFX. The examples and lessons in this book are specifically designed to teach you JavaFX development, as well as the JavaFX scripting language, simultaneously and from the ground up.

### NOTE

The language that JavaFX applications are developed in is called JavaFX Script.

That being said, any experience you have in scripting is going to help you grasp the concepts of JavaFX Script even faster. A basic knowledge of the following concepts, although not necessary, will also help you get up to speed even faster:

- **Java development** JavaFX and Java share more than just their root names. If you have ever written a Java applet—and, more importantly, deployed a Java applet to a web page—you should easily understand the deployment process for JavaFX.
- **HTML** Even though JavaFX can be deployed as standalone desktop applications and as mobile device applications, most people will develop for the Web. One of the great features of the JavaFX development environment is that you will not need to create a single web page to develop for the Web. However, a basic understanding of HTML will help you understand what is going on behind the scenes of your development.
- **Drag and drop** Many things in JavaFX can be created by using the drag-and-drop interface. If you have ever mocked up an application in Visual Basic by dropping objects onto an empty form, you have an advantage in developing JavaFX.

These skills are by no means required, and a lack in any of these areas will not affect your ability to learn JavaFX. Whether you are a seasoned professional developer or a novice who has yet to write your first application, you will be able to easily develop in JavaFX after reading this book. The next section lists the software you will be working with in this book to develop JavaFX.

## Required Software

This section serves as a brief introduction to the software you will be using throughout this book. A few different software elements are used in JavaFX development, and you will be very familiar with these by the end of this chapter. Don't worry if you do not have any of these software elements yet, or have never even heard of one or two of them. By the end of this chapter you will have downloaded and installed all the software required to facilitate JavaFX development.

- **JavaFX SDK** The JavaFX SDK (Software Development Kit) is the major package needed for JavaFX development. The JavaFX SDK contains all the items needed to develop JavaFX applications using JavaFX Script.
- **Java SE JDK** The Java SE JDK (Standard Edition Java Development Kit) is required to compile your JavaFX script into executable code. The JDK is the base for all Java development.
- **NetBeans** NetBeans is the development environment you will use to create your JavaFX apps. Think of NetBeans as a specialized text editor that can use both the JavaFX SDK and the Java SE JDK to compile text into an executable app.

All the pieces of software listed here are free and can be easily downloaded. The next section of this chapter walks you through downloading and installing the required software.

## Downloading and Installing the Required Software

Both NetBeans and JavaFX rely on the Java SE JRE (Standard Edition Java Runtime Environment). Therefore, the Java SE JRE should be installed on your system first. However, the NetBeans 6.9 install not only will automatically install the latest version of the Java SE JRE for you, but will also install the JavaFX SDK. Therefore, you are going to install all your required software at one time by installing NetBeans 6.9.

### NetBeans

NetBeans is an open-source IDE (Integrated Development Environment) that can be used for developing on many different platforms. NetBeans can be used for C/C++, Java, JavaScript, and PHP development, as well as JavaFX. When following the examples in this book, you will do all your JavaFX development within a NetBeans workspace.

The first step is to download the latest version of NetBeans for JavaFX. The latest version of NetBeans can be downloaded from [www.netbeans.org](http://www.netbeans.org).

### CAUTION

The latest version of NetBeans, at the time this book was written, was the NetBeans 6.9 Beta for JavaFX. NetBeans can be downloaded for different languages, and for the purposes of this book you need to download the version of NetBeans 6.9 that is specifically for JavaFX. This will make more sense when you visit the NetBeans download page.

Once you are at the NetBeans download page, the choice of available packages may seem daunting. Fear not, because there is only one flavor of NetBeans 6.9 you need to worry about. You want to download the NetBeans IDE for JavaFX (see Figure 1-1).

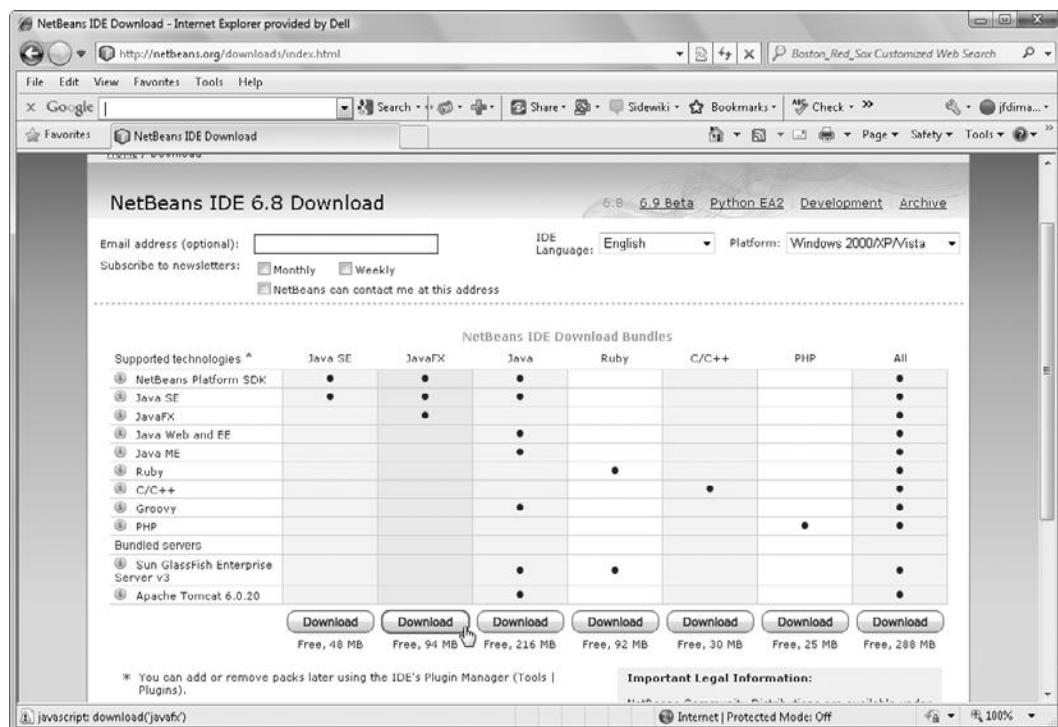


Figure 1-1 NetBeans download page

**CAUTION**

The download from NetBeans labeled "All" is the version of NetBeans for all available technologies. All the examples in this book will still work if you happen to download this version, but you should try to stick with the NetBeans IDE for JavaFX.

Simply follow the installation wizard and you should have no problem successfully preparing NetBeans for development. The installation wizard will recommend default locations for the installation of the NetBeans IDE and the Java JDK; just accept the default locations and the remainder of the installation will be a breeze.

**NOTE**

If Java has never been installed on your computer, you may need to manually install the latest JDK before installing NetBeans.

When the NetBeans installation is complete, the NetBeans IDE should auto-start. If NetBeans does not restart, you may need to bring it up manually. The NetBeans IDE will open to the development start page (see Figure 1-2). The purpose of the NetBeans start page is to offer you tips and news about developing in NetBeans and JavaFX.

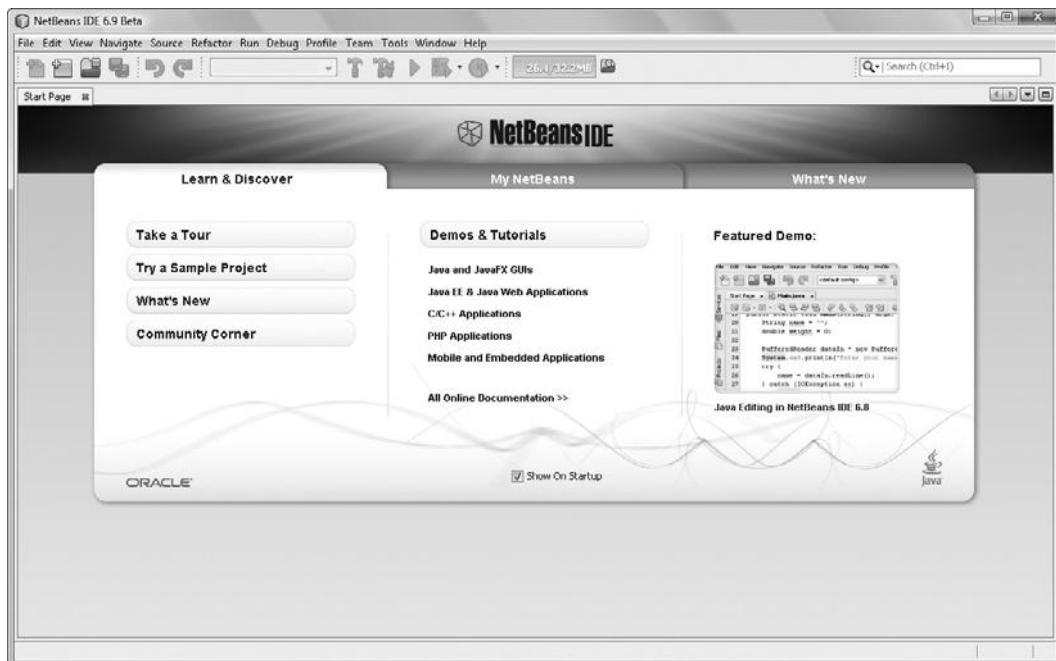


Figure 1-2 The NetBeans default start page

**TIP**

Uncheck the box marked “Show On Startup” to bypass the start page in the future.

At this point, NetBeans is configured and ready for use. The NetBeans installer will ask you to register NetBeans. This step is suggested but not required. Registering your product will give you access to news about upgrades and the NetBeans forums.

## Ask the Expert

**Q: Do I have to use NetBeans for JavaFX development?**

**A:** No, JavaFX can be developed outside of NetBeans. All you really need to write in JavaFX is a simple text editor, the Java JDK, and the JavaFX SDK. However, developing outside of NetBeans would require a fairly good knowledge of command-line compiling in Java.

**Q: Can any other IDEs be used for JavaFX development?**

**A:** Yes, you can also use Eclipse. Eclipse is another open-source IDE that would require the use of a plug-in to work with JavaFX. However, at the time this book was written, no plug-in was available for JavaFX 1.3 and Eclipse.

## Try This Configure Your NetBeans

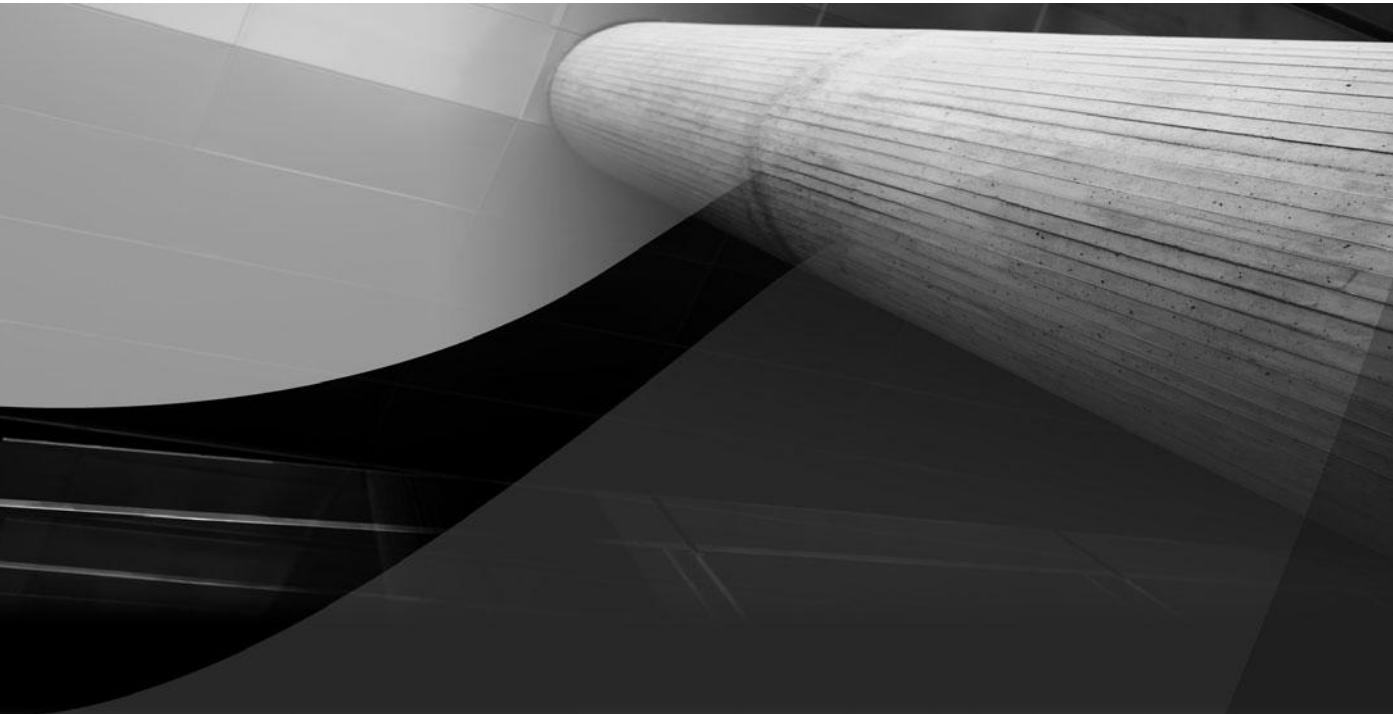
A developer should be comfortable using their IDE. Try to customize the look and feel of your NetBeans IDE to make it a more comfortable place for you to work. When your IDE has a familiar look and feel, you will be much more apt to have creative development sessions.

Open your NetBeans IDE and from the menu bar select Tools | Options. Explore the options provided to you. Experiment with these options by setting different ones and taking note of how they affect the IDE. Find the most comfortable options for you and your method of development.



## Chapter 1 Self Test

1. What is the name of the open-source development environment you will use throughout this book?
2. True or false? You should download the version of NetBeans for All Developers.
3. True or false? The Java JDK will be installed for you automatically if needed (if you have the JRE installed).
4. Which NetBeans settings can you accept the default values for during installation?
5. What is the difference between the JavaFX SDK and the Java JDK?
6. What is the purpose of the NetBeans start page?
7. True or false? You must successfully register NetBeans before using it.
8. At what website is NetBeans available?
9. Name two other applications that closely resemble the functionality of JavaFX.
10. JavaFX will compile for the Desktop, Web, and what other platforms?



# Chapter 2

## Setting the Scene

## Key Skills & Concepts

- Creating a JavaFX project in NetBeans
  - Creating a Stage and a Scene
  - Running a JavaFX application
- 

In this chapter you will learn how to set up a new JavaFX project in NetBeans. JavaFX projects can be confusing for beginners, and sorting through some of that confusion will help you follow the rest of this book. This chapter walks you through, step by step, the process of creating your first project, adding a Stage to the project, adding a Scene to the Stage, and running the application.

## Creating a New JavaFX Project

If you have not already, open your copy of NetBeans 6.9. You will create a new JavaFX project for this chapter using NetBeans.

### **NOTE**

You will use the project created in this section throughout this book. As you progress through the book, you will continue to add script files for the examples in each chapter.

With your NetBeans IDE open, click File | New Project (or press CTRL-SHIFT-N), as seen in Figure 2-1.

Selecting New Project will open the New Project Wizard. Notice that the New Project Wizard contains multiple project categories. This is because NetBeans is used for more than just JavaFX development. However, the Categories option JavaFX and the Projects option JavaFX Script Application should both be pre-selected for you, as seen in Figure 2-2. Accept these defaults and click Next. If these options are not selected, choose them now.

The next step in the wizard is the Name and Location step. NetBeans is looking for a name for your project. Name your project **JavaFXForBeginners**. This is a good, descriptive name for your project that will make it easy to identify.

Finally, unselect the last option (Create Main File) in the Name and Location step. If this option is selected, NetBeans will create your first script file for you. However, NetBeans will add some basic setup code that you want to add yourself this time. Therefore, you will be creating this file separately.

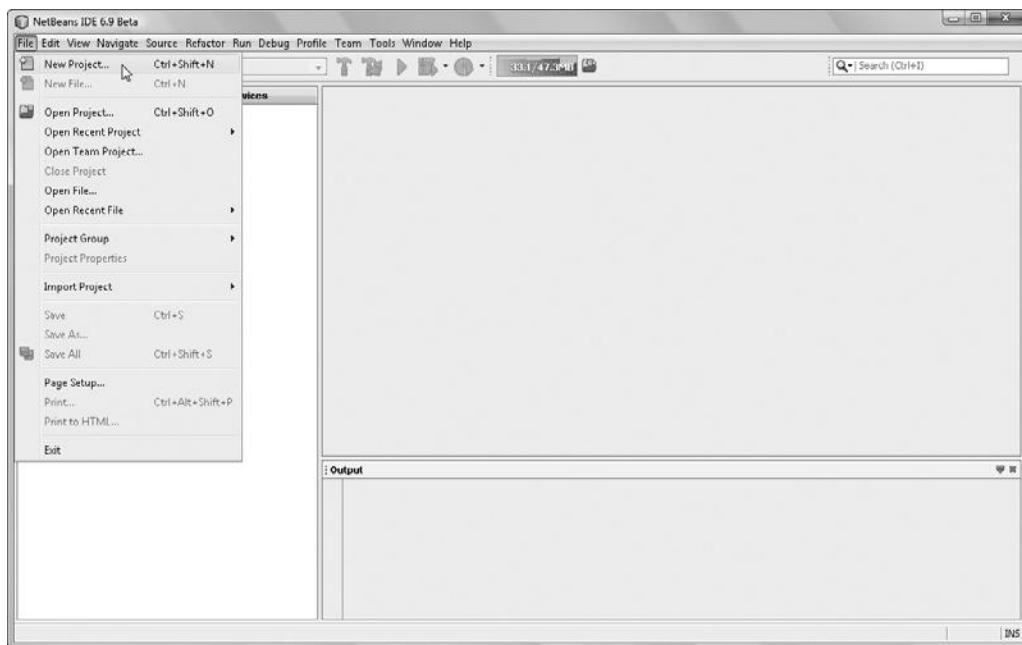


Figure 2-1 Creating a new project

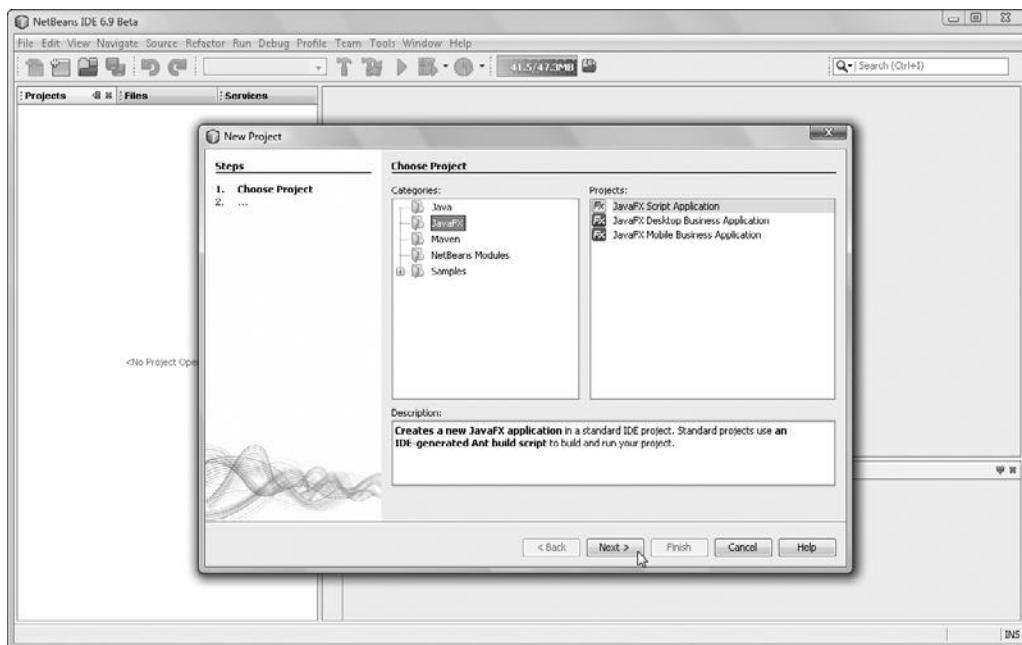
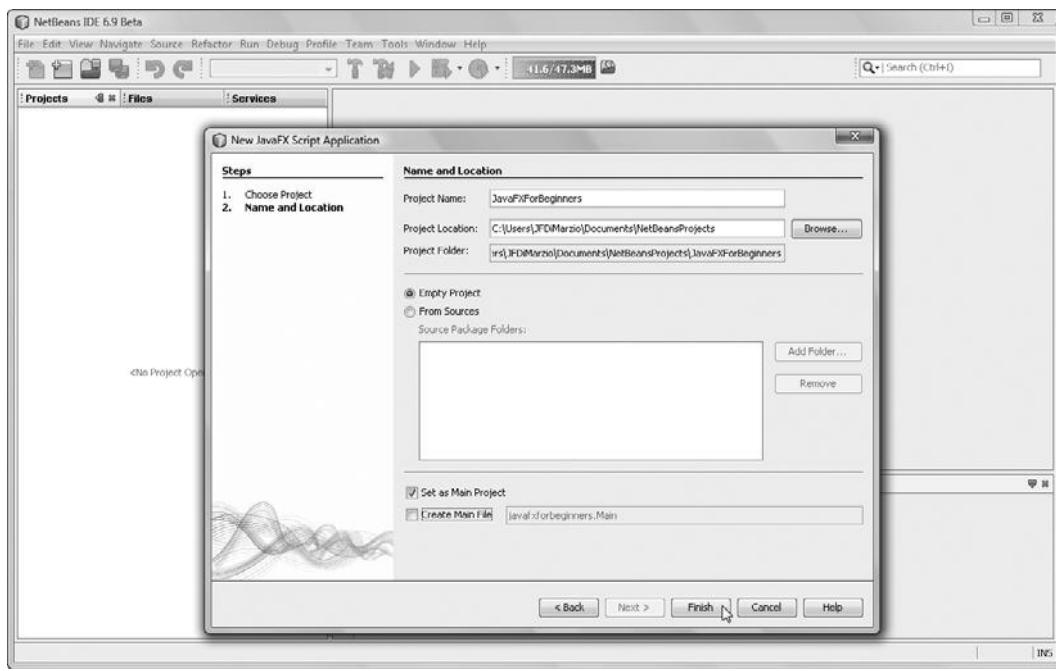


Figure 2-2 New Project Wizard selection window



**Figure 2-3** Name and Location step

Feel free to accept all the other defaults in this step. Your Name and Location step should appear as shown in Figure 2-3.

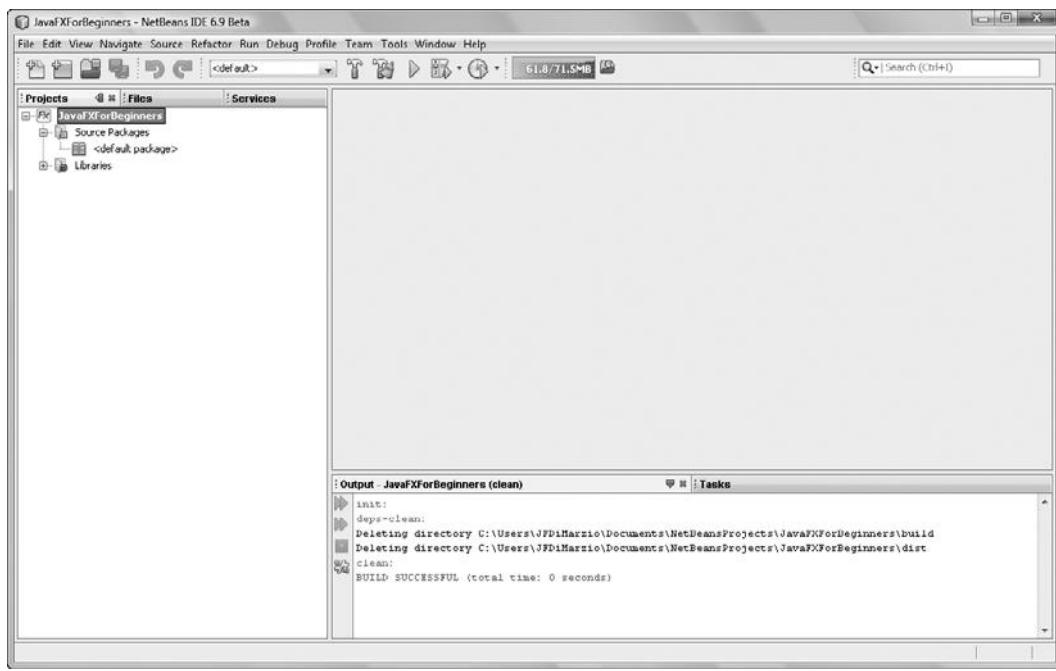
Click the Finish button to create your empty project. In the following section you will add a package and a script to your project.

## The Empty JavaFX Project

Once your new project is created, the New Project Wizard will return you to the NetBeans IDE, as shown in Figure 2-4. On the left side of the screen is a Projects explorer frame. Your JavaFXForBeginners project will be displayed in this frame.

Click the plus sign next to the Source Packages folder. The Source Packages folder will contain the packages for your project.

A Java package is a full collection of classes (or in this case, JavaFX scripts) that are all related. All the files in a package will be compiled together into a JAR (Java Archive) file and can be referenced in other projects. If you have worked with another platform such as Silverlight or .NET, you can think of a Java package as equivalent to a namespace.



**Figure 2-4** NetBeans IDE with a new project

For example, if you were building a set of Java classes that calculate the area of a shape, you could build them into a specific “area calculator” package. This namespace, and all the classes in it, could then be compiled into a JAR file. You could then use that JAR file in any project where you want to be able to calculate area by simply including that JAR file and referencing the namespace.

Packages have a naming convention you will need to adhere to in JavaFX. A package is named using a hierarchical domain structure that represents you as a developer. Much like a website URL in reverse, the namespace name should begin with the top-level domain followed by the related domain names. For this project we will use the following package:

```
com.jfdimarzio.javafxforbeginners
```

This namespace represents the beginners level of my JavaFX instruction (my name is jfdimarzio). Of course, you should feel free to use a namespace that represents you.

**CAUTION**

If you choose to use a package name that better represents you, rather than the example (`com.jfdimarzio.javafxforbeginners`) I will be using in this book, you will need to remember it whenever I refer to the package in new projects.

**TIP**

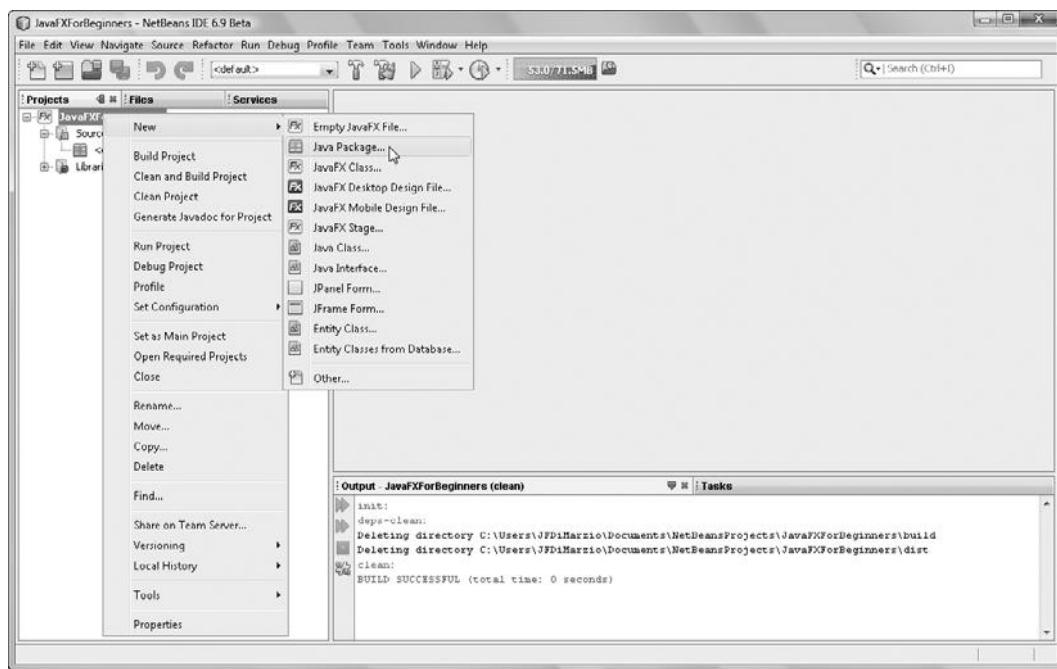
By convention, all Java namespace and project names should be lowercase. For more information about Java naming conventions, visit <http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html>.

If you examine the Source Packages folder of the JavaFXForBeginners project, you will see that you do not yet have a package for your source files (as denoted by the `<default package>` placeholder). Create a new package using the Java naming convention.

To create a new package, right-click the project name in the Projects frame and then select New | Java Package... from the context menu, as shown in Figure 2-5.

**NOTE**

The order of your menu items may differ from those in Figure 2-5.



**Figure 2-5** The Project context menu

NetBeans will give you a default package name of newpackage. You will type over this default package name with **com.jfdimarzio.javafxforbeginners**. You can accept the other default values and click the Finish button to create your package.

You will find that the <default package> placeholder in your project has now become the com.jfdimarzio.javafxforbeginners package. All the files you place in this package will be compiled into the com.jfdimarzio.javafxforbeginners JAR. This is the correct behavior for what you want to achieve.

With your new package created, it is time to add your first script file.

## Adding Working Files to Your Project

Having looked at your newly created project and package, you may be wondering where you begin typing your code. For example, if you were writing a document or memo, you would likely type into a text document file (.txt or .docx). If you were creating a spreadsheet in Microsoft Office, you would type into an Excel file (.xlsx). To create a JavaFX application, you need to type into a JavaFX script file. JavaFX script files have a file extension of .fx.

Take a look at the com.jfdimarzio.javafxforbeginners package; you will be able to quickly determine that, in fact, there are no JavaFX script files for you to begin writing your code. You need to add a file to your package to begin coding. Right-click your package to bring up the context menu. Click New | Empty JavaFX File... to add an .fx file to your package. Name the file **Chapter1** and click the Finish button.

NetBeans has just created your first working script file. The file will be open and displayed in the center of the IDE, as shown in Figure 2-6.

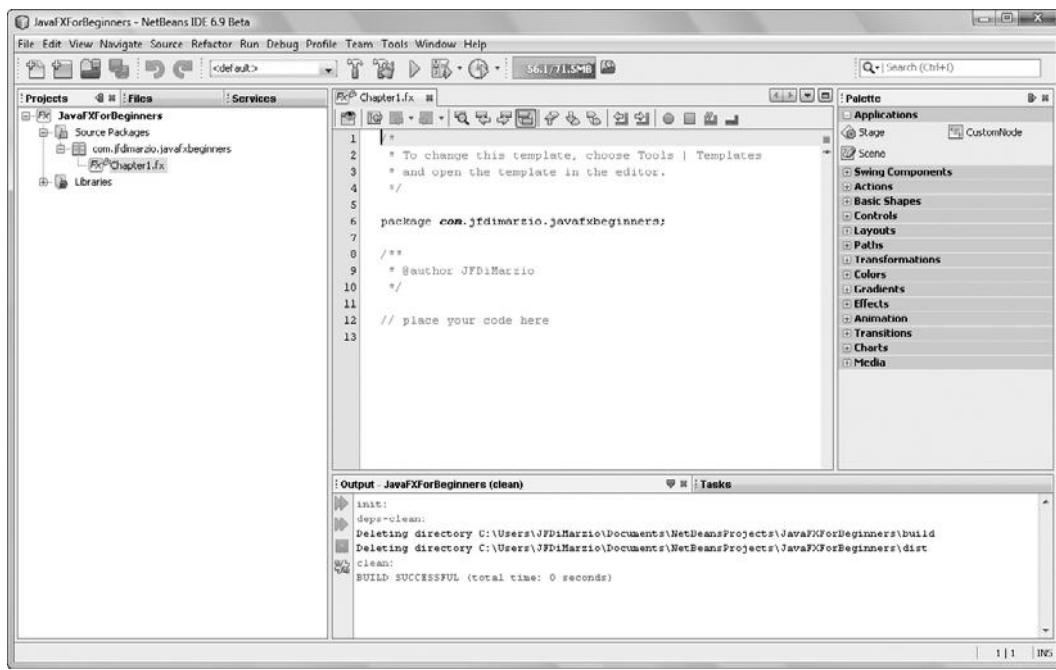
### NOTE

Your screen may differ slightly from that in Figure 2-6.

Your new JavaFX script file should be open in the main panel of the NetBeans IDE. In the following sections, you will take a quick tour of the features available to you in NetBeans as well as the new script file, and you will compile your first JavaFX application.

## Exploring the Empty Project in NetBeans

You should now have a shell of a JavaFX project open in your NetBeans IDE. Your NetBeans JavaFX project should look like Figure 2-6. You might think that an empty project would not be that interesting, and for the most part you are right. However, there are some features and areas of the IDE that you should become familiar with before you begin coding.



**Figure 2-6** Your first script file

For much of your development work in JavaFX, you will be focusing on two areas of the NetBeans IDE. The left side of the NetBeans IDE shows a trio of tabs, labeled Projects, Files, Services. This set of explorers, shown in Figure 2-7, will be your main mechanism for navigating through your projects.

It is not uncommon for projects to start off very small and end up using many files—from code to images and configurations. The explorers help you keep track of these files. They also allow you to move quickly between files, letting you easily work on different files as needed.

One great feature of these explorers in NetBeans is that they allow you to work with multiple projects at the same time. If you have two projects open in the same IDE, you can easily work with them simultaneously without worrying about closing or opening them. This becomes a very handy tool the more you begin to work in NetBeans.

The second area you will become very familiar with by the end of this chapter is the Palette section. The Palette, pictured in Figure 2-8, is located on the right side of the



**Figure 2-7** Projects, Files, and Services explorers



**Figure 2-8** The Palette

---

NetBeans IDE, opposite the explorers. The Palette contains collapsible tabs of code snippets. You will find yourself using these snippets throughout your code, and extensively throughout the early chapters of this book.

A snippet, like those found in the Palette, is a prewritten, reusable piece of code. That is, a snippet is very much like a one of those “fill-in-the-missing-word” comics. In other words, it is a small section of code with a few pieces of key information left for you to provide. These snippets make it extremely easy for anyone to pick up JavaFX for the first time and produce some very functional applications with minimal effort.

The snippets provided for JavaFX are separated logically by function and fall into 15 major categories:

- Applications
- Swing Components
- Actions
- Basic Shapes
- Controls
- Layouts
- Paths
- Transformations
- Colors
- Gradients
- Effects
- Animation
- Transitions
- Charts
- Media

As you progress through this book, you will be introduced to many of the snippets in the Palette. They provide a simple foundation for many of the projects covered in the following chapters. Take some time out to expand each of the Palette categories and explore the snippets included for you.

# Working with the Script File

This section walks you through the empty script file you created in the last section. Believe it or not, even though you didn't write a single line of code, there is a lot you can learn from an "empty" file.

Right now, your file will look similar to this:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package com.jfdimarzio.javafxforbeginners;

/**
 * @author JFDiMarzio
 */

// place your code here
```

The following sections explain the layout and purpose of this empty script file.

## The Comments

The first four lines of code in your "empty" JavaFX Script file are the beginning comments. Comments are not lines of code. In fact, comments are ignored by the compiler. So why bother using them? Comments are added to a file to explain the purpose of the code within the file. They are solely for the benefit of people reading the script files, not the compiler.

### NOTE

Comments can be divided into two types: multiline and single line. Different characters are used to note each.

The beginning comments that have been added to your file read as follows:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
```

By Java coding standards, your beginning comments should include the class's name, version, and creation/modification dates, as well as any copyright information you may want to include.

Take a look at the first two and the last two characters of the comments that were added to your file. The first two characters are `/*` and the last two characters are `*/`. All your multiline comments must begin and end with these characters, respectively. All the other asterisks you see in the comments are added purely for embellishment and readability.

Typically, comments are added to explain the purpose of a script file as a whole, or possibly larger sections of code. For example, place your cursor at the end of the comment line where your name appears and press the `ENTER` key. Highlight the template comments that were added to your file and replace them with the following:

```
/*
 * Chapter1.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 5/1/2010 - created
 *
 * Sample JavaFX script from Chapter 1 of 'JavaFX
 * A Beginner's Guide'
 */
```

You have just added some standard Java beginning comments to your script file. These comments describe your file as being version 1.0 of the file named `Chapter1.fx`. The creation date and a short description (in place of the copyright notice) close out the comments.

In the next section, you will learn about the package statement.

## The package Statement

You should have noticed that the “empty” script file is not very empty. The truth is that NetBeans handles some basic coding necessities for you. After the beginning comments is the following line:

```
package com.jfdimarzio.javafxforbeginners;
```

This line, while having very little direct impact on your code, can teach you some key points about programming in Java. The core purpose of this line is to tell the compiler that all the code under this line belongs to the `com.jfdimarzio.javafxforbeginners` package. The package declaration must be at the top of the script file.

This line of code has three important components you should be aware of. The first word of the line, “`package`,” is a Java keyword. Keywords are special commands that have a predefined meaning to the Java compiler. They are also known as “reserved.” As such, they cannot be used elsewhere in your code as variable names or anything other than their

predefined purpose. The keyword package tells the compiler that anything following the keyword is the name of the package.

The name of your package follows the package keyword (in this case, com.jfdimarzio.javafxforbeginners). Because the purpose of the keyword is to tell the compiler that everything after it is the name of the package, the compiler needs to know where the end of the filename should be. Otherwise, by definition, the compiler would think all the code in the file was the package name. Therefore, Java has a special character set aside as a delimiter: the semicolon (;). The delimiter tells the compiler when it has reached the end of a specific line of code. Although there are very specific exceptions to this rule, you will need to end your lines of code with a semicolon to let the compiler know you have finished a command. If you do not use a semicolon where one is expected, the compiler will give you an error when you attempt to compile.

### TIP

*Compiling* is the process where your code is converted from human-readable text to computer-understandable applications.

The package statement is followed by two more comments. The first, a multiline comment, identifies you as the author of the file. The second is a single-line code that tells you where to begin placing your code.

Notice that single-line comments begin with // rather than /\*. Also, because they are only on one line, there is no need to close them. As such, anything on that line will be considered a comment.

In the final sections of this chapter, you will begin to add your first lines of code to the Chapter1 script file.

## Your First Stage

Most JavaFX nodes go on a Stage. The naming of this node, as the Stage, is very apropos. Think of the Stage as the foundation for the rest of your script's objects. Just as with a dramatic production, the action happens on a stage—and JavaFX is no exception. All of your action is going to take place on a Stage.

### Inserting the Stage Snippet

To add a Stage to your script, focus on the Palette area to the right of the NetBeans IDE. In the Palette area, expand the Applications accordion tab if it is not already expanded.

You will see three application snippets you can add to your script: the Stage, CustomNode, and Scene nodes.

**NOTE**

Delete the single-line placeholder comments before adding a Stage to your file.

Click the Stage and drag it to your script file where the placeholder comments used to be. This will insert a Stage in your script.

Notice that NetBeans has inserted two sections of code into your file. The first section is above your “author” comment. This section of code contains two import statements:

```
import javafx.stage.Stage;
import javafx.scene;
```

The import keyword is another Java keyword that tells the compiler to go to the packages listed and import all the code into this script file. Therefore, in this example, the compiler will get all the code in the javafx.stage.Stage and javafx.scene.Scene packages and place it in this file.

The second section of code that NetBeans added to your file is the Stage node. This section of code creates an empty Stage in your script. Right now, the Stage does not do anything, but it will compile and display an empty window. Your script file now looks like this:

```
/*
 * Chapter1.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 5/1/2010 - created
 *
 * Sample JavaFX script from Chapter 1 of 'JavaFX
 * A Beginner's Guide'
 */

package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;

/**
 * @author JFDiMarzio
 */
```

```
Stage {  
    title : "MyApp"  
    onClose: function () { }  
    scene: Scene {  
        width: 200  
        height: 200  
        content: [ ]  
    }  
}
```

To understand what the code of the Stage does, you must learn some basic JavaFX coding and syntax. The following section explains the basics of JavaFX.

## A JavaFX Script Primer

Now that you have some code in your script file, you can take some time out to look at how JavaFX script is written (that is, its *syntax*). If you have any previous Java or even JavaScript experience, you may be surprised to see that JavaFX Script differs from both these languages slightly.

Each element in the JavaFX script is designated by the *type name* of the node, followed by a set of curly braces. To create a Stage, here is the code required to designate the Stage element:

```
Stage{ }
```

All the attributes or parameters associated with the Stage are placed between the curly braces in a pattern known as *name-value pairs*.

## Name-Value Pairs

Name-value pairs, as the wording suggests, are elements of code written out as the name of the attribute or parameter followed by the value assigned to that attribute. For example, the title of your application, in name-value pair syntax, is written out as follows:

```
title : "MyApp"
```

Change the title of your Stage to **JavaFXExamples**.

You will find as you create more-involved JavaFX scripts that the value of a name-value pair does not have to be as simple as a string (“JavaFXExamples”). In fact, your Stage already contains one complex value.

The Stage has a required attribute called “scene.” The value expected for the scene variable is a Scene element. Look after the title name-value pair in your Stage code and you will see the following name-value pair:

```
scene: Scene{  
    width: 200  
    height: 200  
    content: []  
}
```

In this example, the value for scene is a Scene element. The Scene element has its own name-value pairs for width, height, and content.

Next, you will use a Run Configuration to compile and run your JavaFX script.

## Compiling Your JavaFX Script

To compile your JavaFX script and run it, press the F6 key. You can also click the large green arrow in the NetBeans menu bar. Either of these methods will compile your script into an executable application.

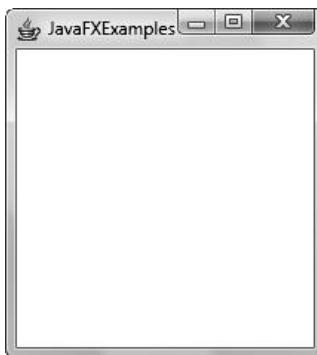
Your script will compile and run based on the selected Run Configuration. The result will be displayed as a separate window.

### **NOTE**

Currently you only have one Run Configuration: <default>. As you progress through the book, you will create others.

In this case, your first JavaFX script will produce a very exciting empty window (shown in Figure 2-9).

In the next chapter, you will create a Hello World application.



---

**Figure 2-9** Your first compiled JavaFX script

## Ask the Expert

**Q:** Do you have to use the snippets to create a Stage?

**A:** No. You do not have to use the snippets for any code creation if you do not want to. You can, with some work, type in all the required code to create any element manually. The pro for using the snippets is that you do not have to remember what attributes, or name-value pairs, are required to create the code. The con for using snippets is that only the required name-value pairs are included. There could be many other optional attributes you will not know about if you only use the snippets.

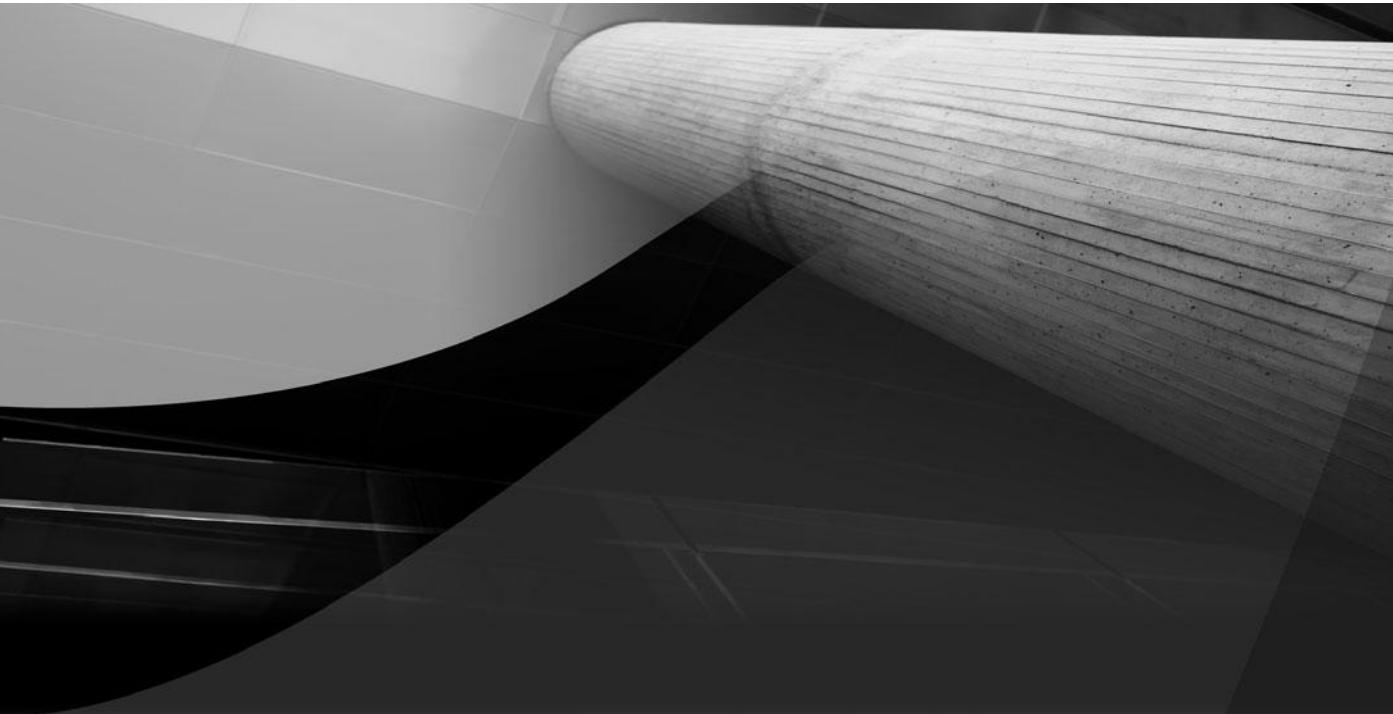


### Chapter 2 Self Test

1. What is the name of the frame where all your projects are listed?
2. What is the name of the wizard used to create a new JavaFX project?
3. What is another name for a namespace?
4. Which panel of the NetBeans IDE lets you navigate through code samples?
5. True or false? The Snippets panel contains predefine pieces of reusable code.
6. What file extension is assigned to JavaFX Script files?
7. What type of word is “package” in the JavaFX script?
8. True or false? Every line of your script must end with a period.
9. What are the beginning and ending characters for comments?
10. What type of variable or attribute is the following?

```
title: "MyApp"
```

*This page intentionally left blank*



# Chapter 3

## Hello World

## Key Skills & Concepts

- Writing text to the screen
  - Understanding profiles
  - Binding values
- 

In this chapter you will start writing JavaFX script. This is where all the prerequisite work from the previous two chapters begins to pay off. All of the time and effort you put into learning the basics in the first two chapters will help you master the JavaFX script nodes introduced in this chapter.

The best part of learning a new language is getting to express your thoughts and visions in a new and exciting way. That is exactly what you will be doing in this chapter. You will learn how to start taking things you may (or may not) know how to do in other languages and bring them to life in JavaFX Script.

## Writing to the Screen

The most common form of human expression is the use of words. The main method of communicating your ideas to others is through words and sentences. We are bombarded with information via words every day. When you are listening to music, driving in your car, surfing the Web, or developing code, you are using and digesting words. Whether the words are spoken or written, like the ones you are reading right now, the best way to get a point across is to use words.

For this reason, the first thing you will learn how to do in this chapter is to write text to the screen using JavaFX Script. Chances are, no matter what type of applications or rich environments you want to create in JavaFX, you will need to write some form of text to the screen. Writing text to the screen is a common, everyday task in development. Therefore, let's start writing text using JavaFX.

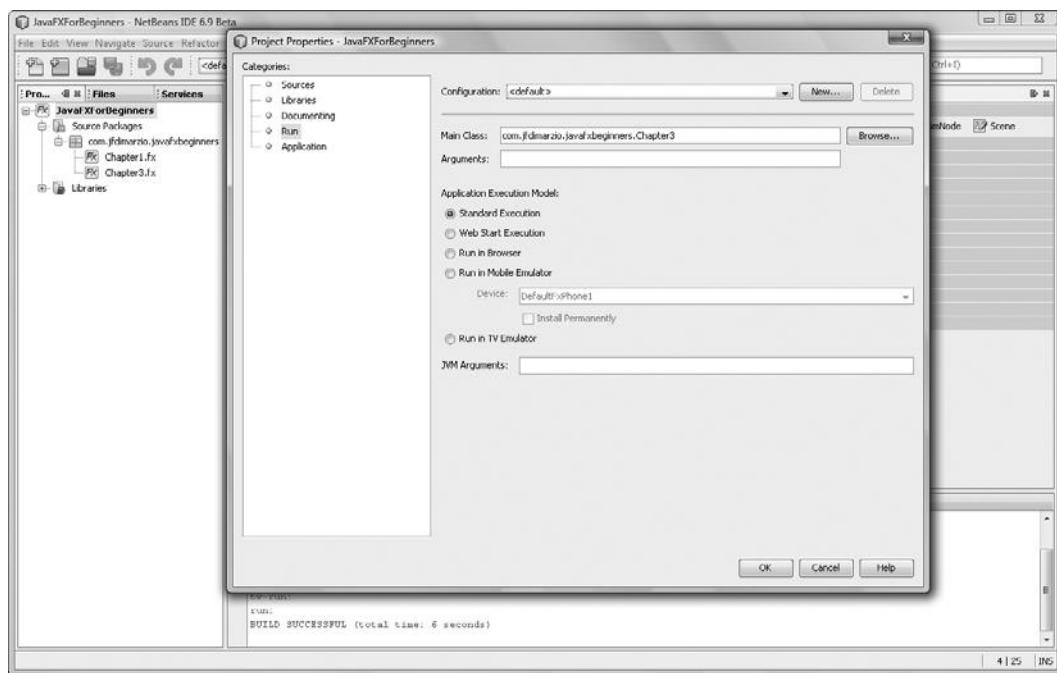
To begin, open NetBeans to the project you created in Chapter 1. Using the steps you have already learned, create a new empty JavaFX file and name it **Chapter3**. We will use this file for all the exercises in this chapter.

**TIP**

For a refresher on creating a new empty JavaFX file, review Chapter 2.

Using the Projects frame, navigate to and open Chapter3.fx (that is, if it does not open automatically for you). In this section you are going to insert some additional comments, create a new Stage and a Scene, and add some text. You are going to discover that with just a few lines of JavaFX Script, you can write anything you want to the screen.

After you add the Chapter3.fx file, you should tell NetBeans that this is your new Main class. Doing so will tell NetBeans to start this file when you compile. To set Chapter3.fx as your Main class, right-click your project name in the Projects frame and select Properties | Run and then set the Main Class: property as shown in Figure 3-1.



**Figure 3-1** Project properties

## Adding Some Descriptive Comments

Your new script file should contain the following code:

```
package com.jfdimarzio.javafxforbeginners;

/***
 * @author JFDiMarzio
 **/
```

Before you begin coding, you should add some descriptive beginning comments, as described in Chapter 2:

```
/*
 * Chapter3.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 5/3/2010 - created
 *
 * First JavaFX Text sample - HelloWorld
 */
```

The comments are now complete. You will easily be able to recognize the purpose of this script's contents in the future simply by reviewing your comments. This will ensure that if you ever need to refer back to any of the code discussed in this chapter, you will be able to locate it with ease.

## Adding the Stage and Scene

You learned about adding Stages and Scenes in Chapter 2. This project will need a Stage, so you need to add one to your Chapter3.fx script for this lesson. Keep in mind that the Stage will hold your Scene, and your Scene will hold the text you want to write to the screen.

The Stage can be added to your script using the Palette. You can insert the Stage by double-clicking the Stage snippet in the Applications section of the Palette. This is the same tool you used in Chapter 2 to insert the sample Stage.

Change the title of your Stage from “MyApp” to “Hello World” (as seen in the following code sample). Changing the title sets this Stage apart from the one used in the last example, and provides it with a more logical reference.

```
Stage {
    title : "Hello World"
    onClose: function () { }
```

```
scene: Scene {  
    width: 200  
    height: 200  
    content: [ ]  
}  
}
```

**TIP**

If you run your JavaFX application as a Desktop application, the “title” you provide for the Stage is also the window title. Therefore, it will be displayed at the top of the window while the window is open.

This code creates a blank Stage named “Hello World” that contains a Scene that is 200 pixels across and 200 pixels high. We are going to leave the dimensions of the Scene at 200×200 for now. You will be changing the dimensions later in this chapter.

Feel free to run your application and wonder at the amazement that is a blank screen. You have several different methods for running your application. For example, you can click the green arrow (on the menu bar), press F6, or select Run | Run Main Project from the menu bar. (Note that clicking the green circle with the arrow on it in the menu bar will run the Chapter2.fx application because it is the last application you ran.)

An intrepid developer may have noticed an option close to all of the “Run” options, labeled “Debug” or “Debug Main Project.” An even more intrepid developer may have clicked Debug and saw that it launches the application like Run does. In fact, Debug is a much more powerful tool than Run, and it allows you step through each line of your code as it runs. For the purposes of this chapter, you should just be using one of the Run options and not worrying about Debug.

Now let’s add some text to this Scene.

## Adding Some Text

Anything you want to display to the screen in JavaFX, for these examples, needs to be placed in the content of your Scene. The Scene node is used to hold and display other nodes. Therefore, to display a Text node (for this chapter’s example), you will need to write the Text node to the Scene’s content.

**TIP**

When discussing the Scene’s content, I am referring to the content attribute of the Scene node.

You are going to use the Text snippet from the Palette to add text to your application. The Text snippet contains the code to create a Text node and four basic attributes that are needed to get you started.

The first step to adding a Text node to your Scene is to place your cursor inside the square brackets—that is, between [ and ]—after the content attribute of the Scene. By placing your cursor here, you are telling NetBeans that this is where you want to insert the Text node. NetBeans will always add snippets to the location of your cursor.

Next, expand the Basic Shapes section of the Palette. This section contains a number of shape nodes, one of which is the Text node. Double-click the Text node to insert the snippet into the script.

Once the snippet has been inserted into your script, you will notice that the value for the font's size attribute is highlighted. NetBeans will auto-highlight any values within a snippet that need to be addressed, as shown in Figure 3-2.

Type the value **26** in the space highlighted by NetBeans. This will change the size of the font used by the Text node. The font attribute takes a Font node as its value. The Font class contains attributes specifically related to the size, style, and color of the font

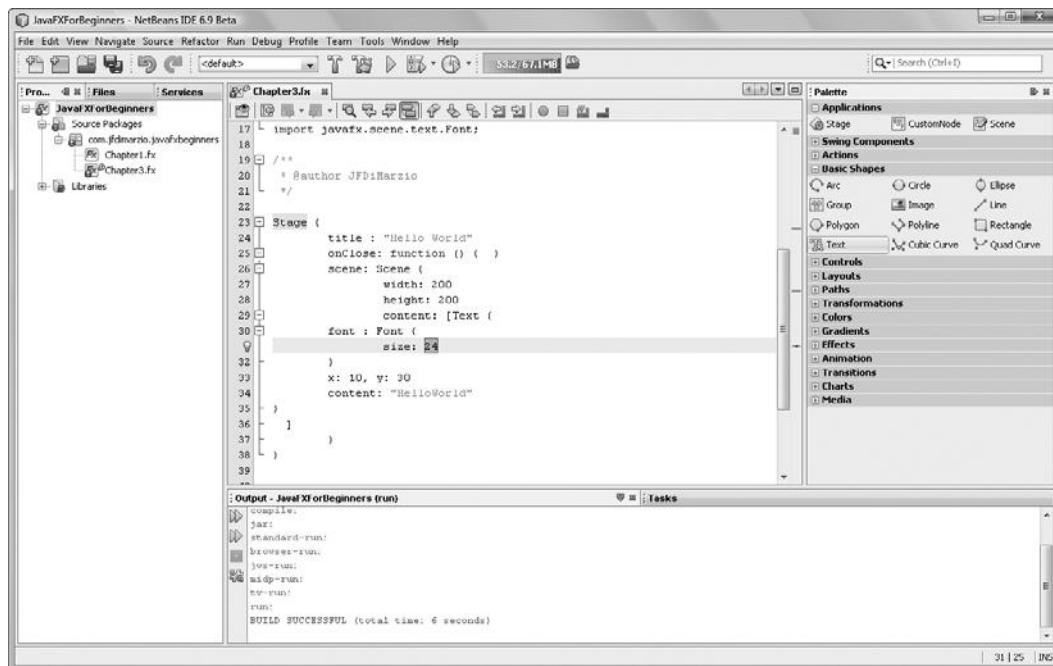


Figure 3-2 Highlighted snippet value

assigned to the Text node. This allows you to have multiple Text nodes in the same application with different fonts. Conversely, it is conceivable that you can assign the same font to multiple Text nodes using advanced programming techniques.

**TIP**

If you were to create and add a Text node manually (without using the Palette menu), you would also need to specify and set the name-value pair attributes manually. One advantage to using the Palette menu is that it forces you to assign values to the more critical attributes, thus ensuring that you have a fully functional node.

After you have entered the value for the font size, press the ENTER key. NetBeans will automatically navigate to, and highlight, the next attribute that needs to be addressed. The values for x and y need to be entered. You can accept the default values of 10 for x and 30 for y by pressing the ENTER key and moving past these values.

The x position and y position indicate the location within the application where you want the text to appear. This position is relative to two points: the top-left corner of the text and the top-left corner of the application window.

**NOTE**

The x and y positions refer to the Cartesian coordinate axes of X and Y. The X axis is the width whereas the Y axis is the height.

Therefore, a position of x10 and y30 would place your text 10 pixels from the left edge of the application window and 30 pixels down from the top. If you are not familiar with the Cartesian coordinate system for user interface layout, do not worry. You will pick up a lot of practice placing objects on your applications using their related x and y coordinates.

**NOTE**

The x and y coordinate values are always written out using a lowercase x and a lowercase y. Do not confuse these with any variables named x or y that you may see in your code.

The final attribute that is supplied for you when using the Text snippet is the content attribute. It contains the actual text you want to display in your application. NetBeans has supplied the default text of “HelloWorld”. Simply press ENTER again to accept this default.

The Text node can accept many more attributes than those listed here. However, these elements represent the minimum of what is needed to create a usable Text node in your application. The Palette forces you to use these four attributes by assigning default values to them and allowing you to change those default values if needed.

With the snippet inserted, your code should look like this:

```
/*
 * Chapter3.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 5/3/2010 - created
 *
 * First JavaFX Text sample - HelloWorld
 *
 */

package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.text.Text;
import javafx.scene.text.Font;

/**
 * @author JFDiMarzio
 */

Stage {
    title : "Hello World"
    onClose: function () { }
    scene: Scene {
        width: 200
        height: 200
        content: [Text {
            font : Font {
                size: 26
            }
            x: 10, y: 30
            content: "HelloWorld"
        }]
    }
}
```

You have now inserted all the code needed to write the default phrase “HelloWorld” to the screen. Read through the preceding code carefully and you will see that the Palette also inserted the necessary import statements to support the new block of code.

### **NOTE**

On its own, the JavaFX script file has no idea what `Text{}` means or what to do with it. Code needs to be imported to tell the compiler what the content type of `Text{}` refers to. Therefore, the import statements that are added by the Insert Template: Text Wizard are necessary for your application to run.

**TIP**

Most nodes you add to a script file will need to be accompanied by a corresponding import statement directing JavaFX to the definition of the node you wish to insert.

Run your project once again. You will see an application window like the one shown in Figure 3-3.

Right now your Run profile is set to <default>. For NetBeans, the default Run configuration executes your code as a Desktop application using the desktop profile. Replace “HelloWorld” with {\_\_PROFILE\_\_} as the Text content value, as follows:

```
Stage {
    title : "Hello World"
    onClose: function () {   }
    scene: Scene {
        width: 200
        height: 200
        content: [Text {
            font : Font {
                size: 26
            }
            x: 10, y: 30
            content: "{__PROFILE__}"
        }]
    }
}
```

**NOTE**

There are two underscores before and after the word PROFILE.



---

**Figure 3-3** Hello World application

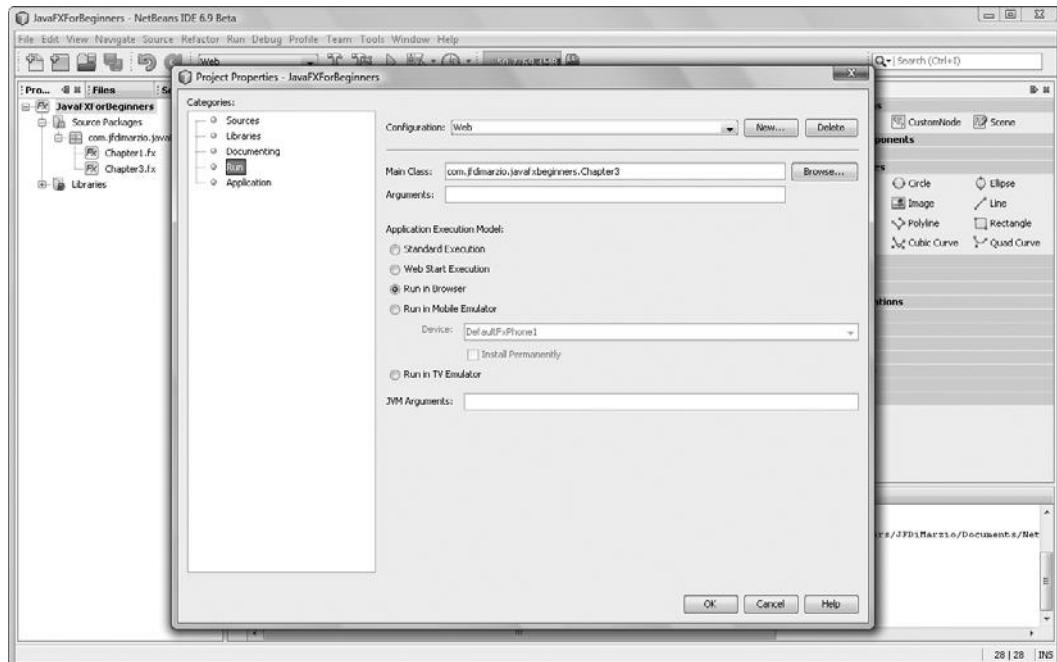


**Figure 3-4** The desktop profile

This will tell JavaFX to print out the name of the profile that the application is running under. Run your script again and you should see the word “desktop”, as shown in Figure 3-4.

JavaFX is capable of running under four different profiles. Let’s modify the Run configuration to see how JavaFX runs.

To change the Run configuration, click the drop-down list to the left of the Run menu bar icon and select Customize. This will open the Project Properties window, as illustrated in Figure 3-5.



**Figure 3-5** Run configurations

In the Project Properties window, click the New button next to the Configuration option. This will create a new Run configuration for you to use. You will name this configuration **Web**.

After you have named the configuration, locate the property section Application Execution Models. You need to select Run in Browser to tell NetBeans to use the browser profile when running your project. Click the OK button to accept your configuration change.

Select the Web Run configuration from the drop-down list and press F6. You will see a window like the one in Figure 3-6.

Notice that the text of the window now reads “browser”. JavaFX is displaying the value of the `{__PROFILE__}` constant, which is now the word *browser*. The JavaFX profile constant can be used to identify which profile your application is running under.

Open the drop-down list once more and select Customize. Create another new configuration named Mobile, as shown in Figure 3-7.

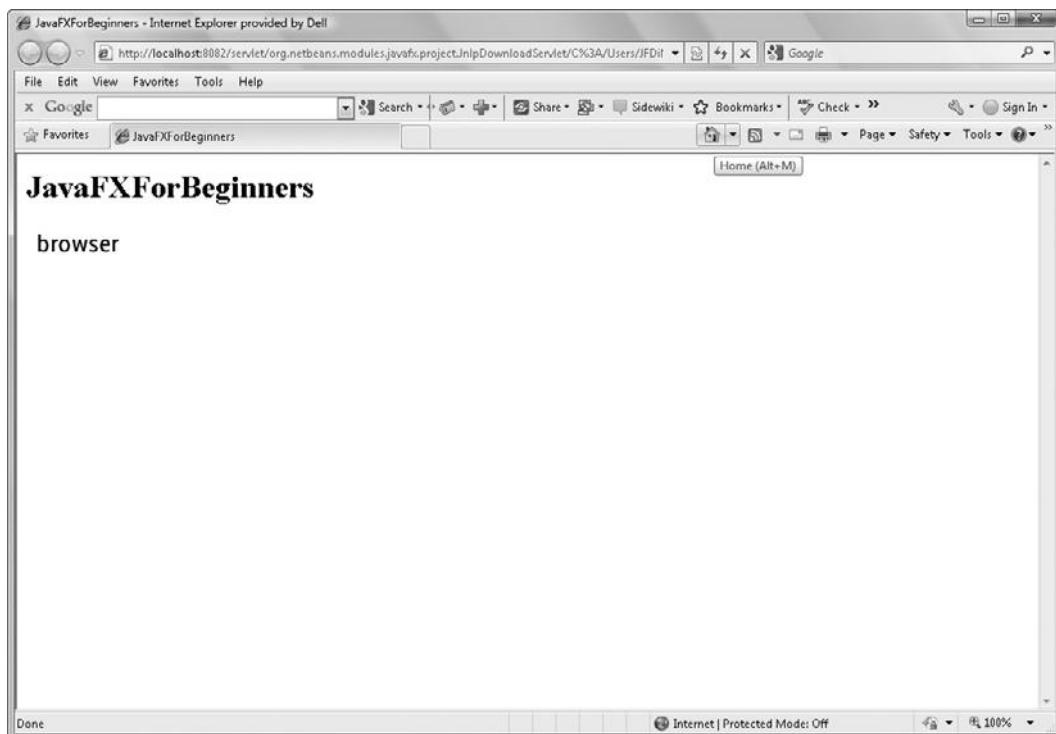
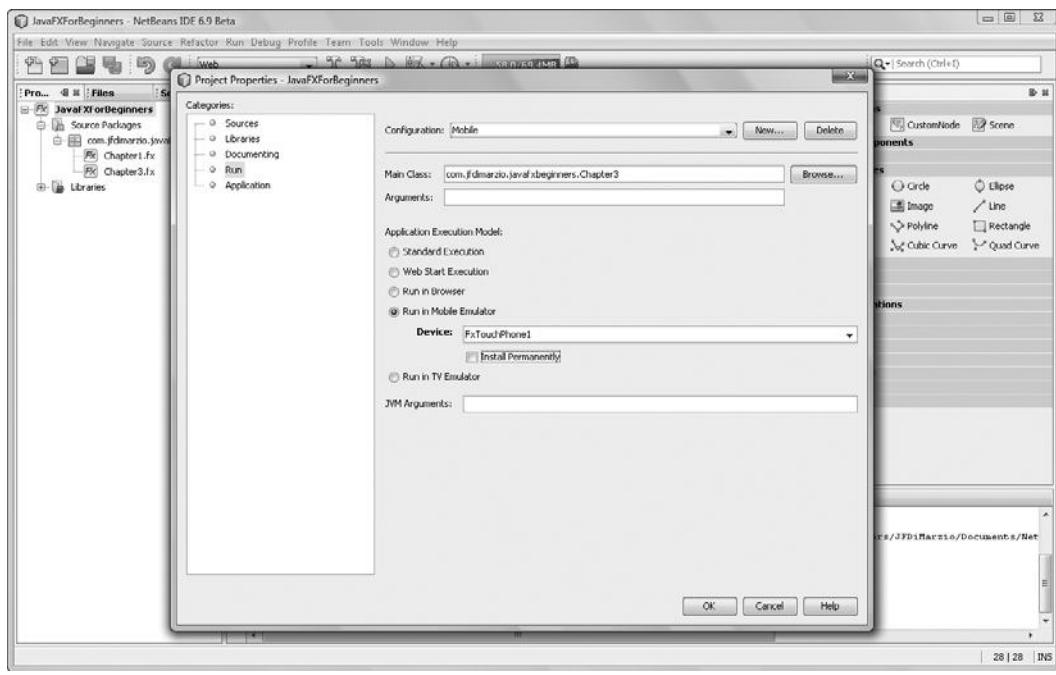


Figure 3-6 Browser profile



**Figure 3-7** Creating a Mobile configuration

Select the Run in Mobile Emulator application execution model and then click the OK button. Run your application again using the Mobile profile. The results are shown in Figure 3-8.

Notice this time that text reads “mobile”. Again, the profile constant has identified which profile you are running under.

## Ask the Expert

**Q:** What is the difference between an application and an applet?

**A:** By its most basic definition, an *application* can run on its own with the support of the Java Virtual Machine. An *applet*, on the other hand, must run within (is hosted by) a browser window. JavaFX is capable of running in both capacities, which is one of its strongest features.



**Figure 3-8** Running in the Mobile Emulator

**Try This**

## Create a TV Run Configuration

Try making the last Run configuration on your own. Using what you have learned in this section, create a Run configuration named TV that runs in the TV Emulator.

When running your application in the mobile phone emulator, you will notice that there is no title bar obstructing the text. Unlike the applet, where a band of text remnants was visible “floating” on the page, the text on the mobile phone emulator is only obstructed by the top of the phone’s screen.

It is important for you to explore the differences between the Run configurations available to you and your applications. Looking at how all three of these Run configurations display your text (and other nodes) will allow you to alter your scripts as needed and create the best rich environments for your users.

You should now congratulate yourself. You have just mastered one of the two basics skills in learning JavaFX. You possess the root knowledge needed to place a node in your script and display it to the application window. Although on the surface it may seem like you just displayed some simple text to the screen, the process you followed in doing so will be used throughout this book when working with most other JavaFX nodes.

In the next section of this chapter you will learn how to add functions to your scripts. These functions will help you change the behavior of your node attributes.

## Adding a Function

After having learned about adding nodes to your script, you may be left wondering about how these simple nodes can be used to make rich interactive environments. The truth is that nodes such as Text and Scene are just a very small part of the overall JavaFX experience.

One way you can add some power to your JavaFX applications is by using functions. Functions are defined blocks of code that can perform a complete task and pass a result to the calling code. The advantage to using functions in your scripts is that they give you more power and more control over the capabilities of your application.

### A Script Function Primer

A Script function is defined outside of the class definition and can be called anywhere inside your script. This means you can create a function to perform a specific task and call it from anyplace inside your script file, any number of times.

Think of a Script function as a tech support call center, and you are a script file. You can call the tech support call center to perform a task that you do not have the capability to perform on your own. The call center takes the call, does some work, and returns you an answer. You then hang up the call and go on with your own duties. Script functions perform the same way.

Script functions are defined using the `function` keyword, and the syntax for creating them is as follows:

```
function <function name> ( <function parameters> )
<: return type> { <function code> }
```

The first, and most important, item is the `function` keyword. This keyword tells the compiler that everything between the curly braces—that is, between `{` and `}`—is a function and should be compiled as such. The compiler will put the function code together in a way that allows other parts of the script to call it and use the returned values.

```
function {}
```

Next, `<function name>` is the name you assign to the function to separate it from other functions in your script. When naming your function, try to use action words that describe the purpose of the function, such as `paintCar`, `addValues`, or `getTextFromBox`. Notice that the first letter is always lowercase. The first letter of each remaining word in the function name is capitalized. This style of naming is known as camel casing.

### NOTE

The act of camel casing your function names is not enforced by the compiler. Therefore, you could name a function `addValues` or `AddValues` and the JavaFX compiler would not stop you. However, it is a convention that is followed by most Java developers. This would be a good habit to begin, and it will make your code look more professional.

In some situations, the name of the function is not needed. One such situation is when you are creating the function specifically to be used as the value to one specific attribute of a node. You will explore this situation later in the book, so for now assume that all functions have names.

```
function addValues() {}
```

Following the function name is a list of any parameters that can be passed into the function. For example, a function named `addValues` would be useless without values to add together. You can create input parameters for your function that allow you to send values to the function. The function can then use these values in its code body. The example that follows creates two input parameters for the `addValues` function:

**NOTE**

All input parameters should be placed within the parentheses that follow the function name and separated by a comma.

```
function addValues(valueA, valueB) { }
```

A return value type can be placed after the parentheses that follow the function name. The return type tells the compiler that when this function is finished running it is going to return this type of value back to the code that called it. For example, a function that adds two values together and produces a result should return that result as a double:

```
function addValues(valueA, valueB) : Double { }
```

**NOTE**

There are times when the return type of your function would have no corresponding type, or when your function may not need to return anything at all. In these instances it is a good practice to declare the function type as : void. The function will still work if you leave the type designator off completely—it is just good practice to type it as void.

Finally, the code or body of the function should be placed between the curly braces. This is the meat of the function, and it does all the work of the function. If your function returns a specific value, the return keyword is used to push that value back to the calling code:

```
function addValues(valueA, valueB) : Double {  
    return (valueA + valueB);  
}
```

**NOTE**

Functions should not be followed by a delimiting semicolon (;), but the code lines within the body of the function should.

Now that you have explored the process of writing a function, you will create one that can be used to write a line of text to the screen. The function will be named sayHello and will return a String type. Write the function so that no input parameters are needed, and be sure to remember the opening and closing curly braces.

Looking at your current script, begin writing your function just above the Stage. If you have started writing the function correctly, your script should look like this:

```
/*  
 * Chapter3.fx  
 *  
 * v1.0 - J. F. DiMarzio  
 */
```

```
* 5/3/2010 - created
*
* First JavaFX Text sample - HelloWorld
*
*/
package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.text.Text;
import javafx.scene.text.Font;

/**
 * @author JFDiMarzio
 */

function sayHello() :String{
    return "";
}

Stage {
    title : "Hello World"
    onClose: function () { }
    scene: Scene {
        width: 200
        height: 200
        content: [
            Text {
                font : Font {
                    size: 26
                }
                x: 10, y: 30
                content: {"__PROFILE__"}
            }
        ]
    }
}
```

### NOTE

Once you have assigned a return type to a function, you must return a value of that type. Because you have not completed the function in this example yet, an empty string ("") is returned.

Because you have established that the return value of the function is a String, you need to write the body of the function to pass some kind of text back to the caller. In this case,

use the string “Hello – from a function.” The easiest way to accomplish this is by using the return keyword followed by the string of text you want the function to pass:

```
function sayHello() :String{  
    return "Hello - from a function.";  
}
```

Your function is now written. Anything that calls this function will be passed the phrase “Hello – from a function.” This string can be used in other parts of your script, as needed. For example, you can use this function to pass this string of text to your Text node to display in the application window.

The Text node needs to be altered to call the sayHello() function. The logical place to make this modification is the content attribute of the Text node. Currently, the content attribute of your text node appears as follows:

```
Text {  
    font : Font {  
        size: 26}  
    x: 10, y: 30  
    content: {__PROFILE__}  
}
```

Notice that you are currently directly specifying the content attribute using the value of the {\_\_PROFILE\_\_} constant. You can replace this with a call to your newly created function. Because the function returns a String type, the content attribute will use the return value as though it were a directly specified string:

```
Text {  
    font : Font {  
        size: 26}  
    x: 10, y: 30  
    content: sayHello();  
}
```

Your full script should appear as follows:

```
/*  
 * Chapter3.fx  
 *  
 * v1.0 - J. F. DiMarzio  
 *  
 * 5/3/2010 - created  
 *  
 * First JavaFX Text sample - HelloWorld  
 */
```

```
package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.text.Text;
import javafx.scene.text.Font;

/**
 * @author JFDiMarzio
 */

function sayHello() :String{
    return "Hello - from a function.";
}

Stage {
    title : "Hello World"
    onClose: function () {  }
    scene: Scene {
        width: 200
        height: 200
        content: [
            Text {
                font : Font {
                    size: 26
                }
                x: 10, y: 30
                content: sayHello();
            }
        ]
    }
}
```

Using the default Run configuration, run your application. As shown in Figure 3-9, you will see the text “Hello – from a function.”



**Figure 3-9** Hello – from a function.

**NOTE**

You may need to expand the JavaFX application window to see the full text.

In this section you have learned how to write a Text node to the screen. You have also learned about functions and how to use them to return a String type to the content of a Text node. In the next section of this chapter you will learn about one more way to display text using a function.

## Using bind with a Text Node

JavaFX includes a very useful keyword called “bind.” The bind keyword is used, on a basic level, to associate one value with another. However, on a more powerful level, bind is used to link one attribute value with another value—even if the second value changes.

Using bind can be a very powerful asset for you because you can bind an attribute value to a variable, change the value of that variable, and those changes will automatically be passed on to the attribute that is bound to the variable. Let’s discover how to use bind with a Text node.

### Creating Variables

You have two ways to define variables in JavaFX: using the *var* keyword and using the *def* keyword. The var keyword is used to define the majority of the variables you will use in JavaFX. Variables defined using the var keyword can be written to and read from as you see fit. The def keyword also allows you to define variables that are written to and read from. The difference here is that when you define a variable using the def keyword, it is understood that you will assign that variable an initial value and never change it.

The following code defines a variable named *helloWorld*:

```
var helloWorld = "";
```

**NOTE**

JavaFX variables also follow a naming convention. Although not enforced by the compiler, variables should be camel cased. Also, although the characters \$ and \_ are allowed as the first character of a variable name, they should generally be avoided.

You have now defined a variable named *helloWorld*. This is all that is needed to begin using the variable. However, the code could use a little tweaking. Notice that we have not told JavaFX what type of variable *helloWorld* should be. This is perfectly legal because JavaFX will try to determine what type of variable you need based on the data assigned to it. If you assign a 3 to the variable, JavaFX will infer that your variable should be typed

as an integer. If you assign a value of “Hello” to the variable, JavaFX will infer that your variable should be a String type.

Letting JavaFX infer what type your variable should be can be useful, but it does use extra resources that would not be used if you type your variable to begin with. Also, if your script is more complex and relies on more complex types, variable type inference could be hard to impossible, and may throw errors. Therefore, because you already know your helloWorld variable is going to be a String type, take the time out to type the variable correctly, like this:

```
var helloWorld :String = "";
```

Create a small function that assigns the string “Hello – From bind.” to your helloWorld variable. You can name the new function sayHelloFromBind:

```
function sayHelloFromBind() {
    helloWorld = "Hello - From bind.";
}
```

You should remove the existing sayHello() function from your script and replace it with your new variable definition and sayHelloFromBind() function. Your script should appear as follows:

```
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.text.Text;
import javafx.scene.text.Font;

var helloWorld :String = "";
function sayHelloFromBind(){
    helloWorld = "Hello - From bind.";
}

Stage {
    title : "Hello World"
    onClose: function () { }
    scene: Scene {
        width: 200
        height: 200
        content: [
            Text {
                font : Font {
                    size: 26}
                x: 10, y: 30
                content: "";
            }
        ]
    }
}
```

With your variable defined and your function created, it is time to bind the Text node content attribute to the helloWorld variable.

## Binding to helloWorld

Binding to your helloWorld variable is easy. To bind to any variable, use the bind keyword followed by the name of the variable you want to bind to. The bind keyword followed by the variable name should be used as the value for the attribute you want to bind to. Take a look at the following code fragment to understand what binding should look like:

```
content: bind helloWorld;
```

This is all that is required to bind a value to variable. The value that is assigned to helloWorld from the sayHelloFromBind() function will be displayed on the screen as soon as the sayHelloFromBind() function is called.

The following line of code is used to call the sayHelloFromBind() function:

```
sayHelloFromBind();
```

Your script should appear as shown here:

```
/*
 * Chapter3.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 5/3/2010 - created
 *
 * First JavaFX Text sample - HelloWorld
 *
 */

package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.text.Text;
import javafx.scene.text.Font;

/**
 * @author JFDiMarzio
 */
```

```
var helloWorld :String = "";
function sayHelloFromBind(){
    helloWorld = "Hello - From bind.";
}
sayHelloFromBind();
Stage {
    title : "Hello World"
    onClose: function () {   }
    scene: Scene {
        width: 200
        height: 200
        content: [
            Text {
                font : Font {
                    size: 26
                }
                x: 10, y: 30
                content: bind helloWorld;
            }
        ]
    }
}
```

If you follow the code logically, you will see that first the variable, helloWorld, is set to a blank string (""). Then the sayHelloFromBind() function is called, changing the value of helloWorld from a blank string to “Hello – From bind.” Now run the application and view the results. The results should appear as in Figure 3-10.

In the next chapter, you will begin to draw shapes to the screen.

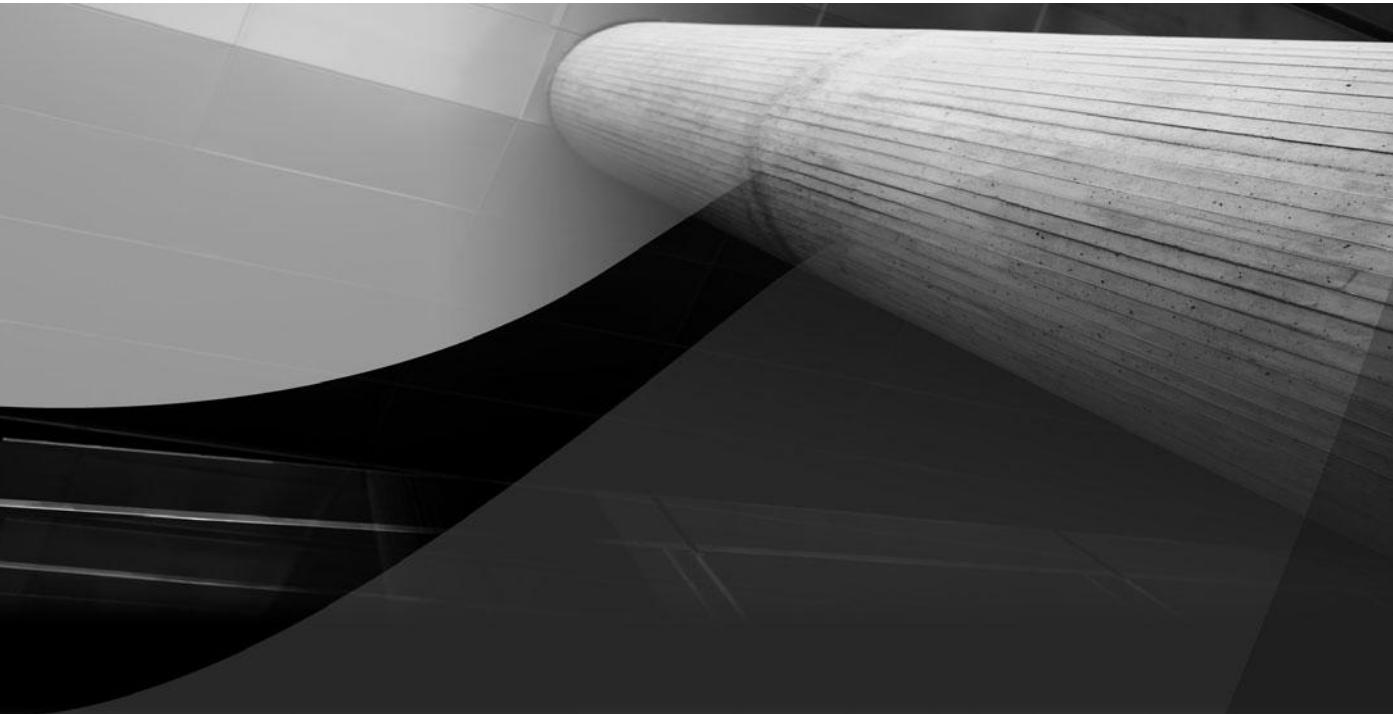


**Figure 3-10** Hello using bind



## Chapter 3 Self Test

1. What are the four basic attributes that need to be defined for a Text node?
2. In what Palette menu is the Text node located?
3. What Run configuration is used to run your script as a Desktop application?
4. When creating a function, where would you specify input parameters?
5. True or false? The function name MyFunction follows proper naming convention.
6. What keyword is used to return a value from a function?
7. Name the two keywords that can be used to create variables.
8. How would you type a variable as string?
9. True or false? The bind keyword is used to bind a variable and make sure it never changes.
10. What syntax would be used to bind a variable named tooMuchText to a content attribute?



# Chapter 4

## Creating Shapes

## Key Skills & Concepts

- Drawing lines to the screen
  - Working with the context menu
  - Creating complex shapes
- 

In this chapter you will learn how to draw shapes to the screen. Shapes add interest and dimension to your applications. You will start by learning how to draw a basic line. From there you will move on to a polyline and then to a rectilinear shape. From these basic rectilinear shapes, you will move on to polygons and curvilinear shapes.

### **NOTE**

If you are interested in game development or other animation-based media, polygons are very important. Polygons create the base of most 3-D objects.

## Drawing Shapes

You will be drawing shapes on your application, in this chapter, without the help of the Snippets menu. The Snippets are very good tools, and can be extremely handy when you need some quick code. The problem is that if you rely too heavily on this automatically produced code, you will not learn how to create these elements on your own. The goal in learning a new language is to gain the knowledge needed to write code. However, if you only learn to rely on automated Snippets, you really will not learn that much.

In the following six sections, you will begin drawing lines to the screen. As you move through these sections, you will start to create more complicated shapes, including polygons and ellipses. Finally, you will draw pre-rendered images to the screen before moving on to applying effects.

## Before You Begin

To begin, you need to prepare your project. All the lessons in this chapter can be done in a single script file. Create a new script file (using the empty JavaFX File template) named **Chapter4.fx**. Using the lessons covered in Chapter 2, create a Stage and a Scene.

**NOTE**

You will always want to set the files for these chapters as the main class for the project. To do this, right-click the Project and go to Properties | Run Properties and set the current file, Chapter4.fx, as the main class.

Before continuing with this chapter, ensure the code in your Chapter4.fx file looks like this:

```
/*
 * Chapter4.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 5/11/2010 - created
 *
 * Creating basic shapes
 *
 */
package com.jfdimarzio.javafxforbeginners;
import javafx.stage.Stage;
import javafx.scene.Scene;

/**
 * @author JFDiMarzio
 */
Stage {
    title : "Basic Shapes"
    onClose: function () { }
    scene: Scene {
        width: 200
        height: 200
        content: [ ]
    }
}
```

## Lines and Polylines

A line is the most basic form of a shape. Almost any shape you can think of, draw, or create is made up of one or more lines. From squares and triangles, as collections of straight lines, to circles and ellipses, as collections of curved lines, understanding the basics of line drawing is crucial.

Given that almost all other shapes you will draw are based on the line, it seems logical to begin this chapter learning about lines.

What is a line? This may seem like an elementary question. We have all been drawing lines on paper, with pencils, crayons, and markers, since our earliest school days. You can use that experience of drawing lines on paper with a crayon to understand what a “line” is to JavaFX.

When you draw a line on a piece of paper, you place your crayon down on one point, drag it across to another point, and lift up the crayon. The line you draw is as thick as the crayon you used to draw it. The line is also the same color as the crayon. This same logic can be applied to drawing a line in JavaFX.

When you draw a line in JavaFX, you must tell the compiler the start point and end point of your line. The values you need to specify are startX and startY as well as endX and endY. The compiler will be able to take this information and draw a line on the screen between these two points.

### **NOTE**

The start and end points are expressed using the Cartesian coordinate system explained in Chapter 3.

You must start by specifying an import statement in your script. The import statement you need contains the code for drawing lines. The package for lines is required for the compiler to understand what your script is trying to do, and to draw a line accordingly.

At the top of your Chapter4.fx script, under the existing import statements, include the following statement:

```
import javafx.scene.shape.Line;
```

You will learn two different shortcuts now for drawing the line. First, place your cursor within the brackets—that is, between [ and ]—that follow the Scene content attribute in your script. Type in the word **Line** between the brackets, followed by opening and closing curly braces ({}). Place your cursor between the curly braces. Now you are going to bring up the context menu to help you do the rest.

With the cursor still in between the curly braces, press the CRTL-SPACEBAR. This keystroke shortcut will bring up the context menu shown in Figure 4-1.

The context menu is an invaluable tool that shows you all the options available to you from a specific place in your script. The context menu can be used to discover elements or attributes that you otherwise would not be aware of. The context menu can even be used to see some of the values that can be assigned to attributes.

When you call the context menu after an element name, but before the element’s curly braces, the menu will show you the structure of the element. If you call the context menu

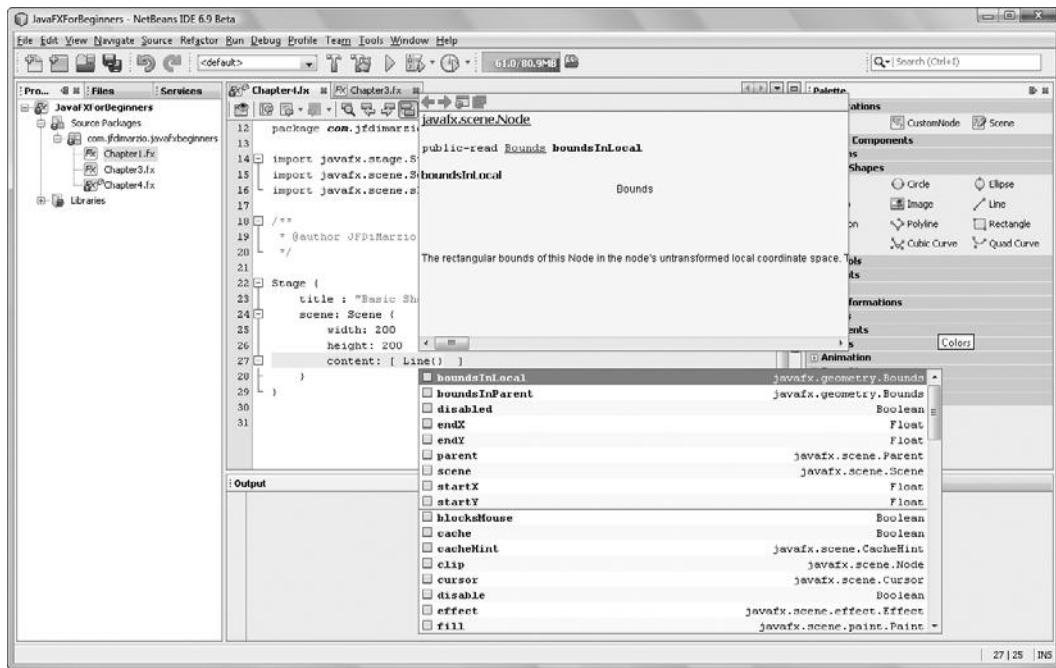


Figure 4-1 Context menu

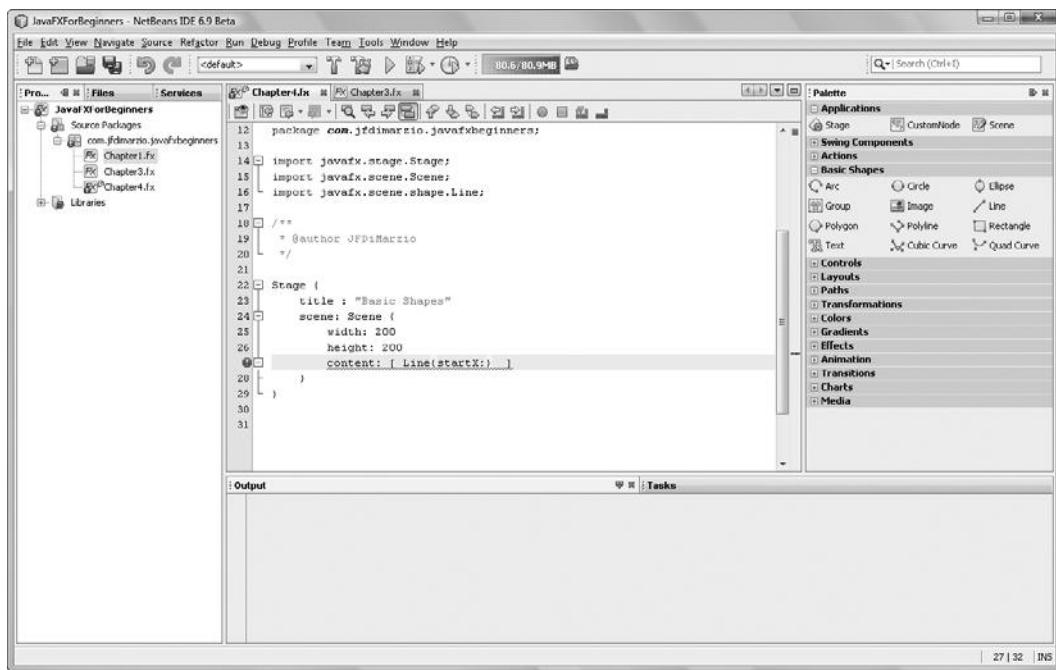
from within the curly braces of an element, it will show you all the available attributes for that element. You should commit the CRTL-SPACEBAR keystroke shortcut to memory because you will find yourself using it quite often.

If you press the ENTER key while the context menu is on an attribute, as shown in Figure 4-1, that attribute will be inserted into your script.

Notice that in your context menu, one of the attributes you can add is startX. The startX attribute is one of the four attributes you need to specify to draw the line. Make sure the startX attribute is highlighted and press the ENTER key.

The context menu should have inserted in startX attribute and a colon (:), as shown in Figure 4-2.

Assign the startX attribute a value of 10. Keep in mind that all the attributes for an element are separated by a comma, semicolon, or nothing. The fact is that JavaFX is very forgiving when it comes to attribute delimiters. With that said, try to stick with using a semicolon, just to be standardized. Therefore, insert a semicolon after the startX value



**Figure 4-2** Adding an attribute with the context menu

and bring up the context menu again. Use the context menu to insert startY and assign it a value of 10 as well.

You will be drawing a line that extends from x10, y10 to x150, y150. Seeing as you have just assigned 10,10 to startX and startY, use the context menu to finish assigning the correct values to endX and endY. Your finished script should look like this:

```
package com.jfdimarzio.javafxforbeginners;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.shape.Line;

/**
 * @author JFDiMarzio
 */
Stage {
    title : "Basic Shapes"
    scene: Scene {
        width: 200
        height: 200
        content: [ Line(startX:,
            startY: 10,
            endX: 150,
            endY: 150)
    ]
}
```

```
        content: [ Line {startX : 10;
                      startY : 10;
                      endX : 150;
                      endY : 150;
                    }
                  ]
    }
```

Run your script, and you should get a line that looks like the one shown in Figure 4-3.

This looks like a great line, and you should be very proud of the job you have done—but all things considered, it is not very exciting. You can assign a few more attributes to this line to make it a little more interesting. The discussion that follows details all the attributes available to the Line element.

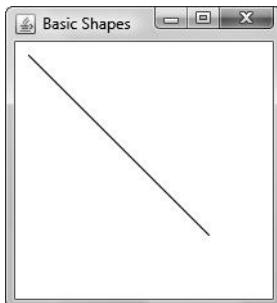
To begin, to make your line a little thicker, add an attribute to your Line element called `strokeWidth`. The `strokeWidth` attribute does exactly what the name implies—it governs the thickness or width of the Line element. Assign the `strokeWidth` attribute a value of 15. This will make your line stand out a bit more.

Adding some color to your line will make it pop out even more. Color is a common attribute for shapes. To work with color, you need to import the `javafx.scene.paint.Color` package. Therefore, add the following import statement to your script:

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

To color the Line element a nice shade of tomato red, create a `stroke` attribute and assign it a value of `Color.TOMATO`. The `stroke` attribute acts just like the stroke of a paint brush. It can contain information about color, gradients, and other visually appealing details of a shape. Here's the code for the `stroke` attribute:

```
stroke : Color.TOMATO
```



**Figure 4-3** A line

Your new Line element should have the following attributes and values:

```
Line {startX : 10;
      startY : 10;
      endX : 150;
      endY : 150;
      strokeWidth: 15;
      stroke: Color.TOMATO
}
```

Run your script, and you will see a nice red line from the upper left to the lower right of the screen.

Let's draw three of these lines to form the shape of the letter *U*. Try to lay out the three lines on your own. The first line should run down the left side of the screen. The second line should run from end of the first line, across the bottom, to the right side of the screen. Finally, the third line should run from the end of the bottom line to the top on the right side of the screen. When you are finished, compare your code to the code that follows:

```
package com.jfdimarzio.javafxforbeginners;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.shape.Line;
import javafx.scene.paint.Color;

/**
 * @author JFDiMarzio
 */
Stage {
    title : "Basic Shapes"
    scene: Scene {
        width: 200
        height: 200
        content: [Line {
            startX : 10;
            startY : 10;
            endX : 10;
            endY : 100;
            strokeWidth : 15
            stroke: Color.TOMATO;
        }
        Line {
            startX : 10;
            startY : 100;
            endX : 100;
            endY : 100;
        }
    }
}
```

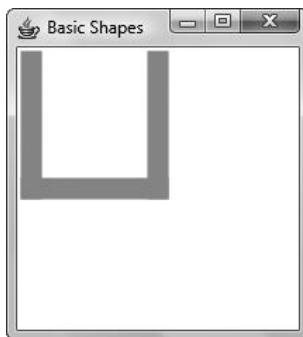
```
        strokeWidth : 15;
        stroke: Color.TOMATO;
    }
    Line {
        startX : 100;
        startY : 100;
        endX : 100;
        endY : 10;
        strokeWidth : 15
        stroke: Color.TOMATO;
    }
}
}
```

Run your code and compare the results to those in Figure 4-4.

Drawing separate lines for every shape and configuration you can think of is not a very economic use of your time as a developer. Just imagine what it would be like if you had to draw every line, one at a time, to form simple shapes. Luckily there is another type of line called a polyline. It lets you specify more than one point in one line, the last of which is the end point. Specifying more than one end point lets you draw more complex lines with less code. Let's try to draw the same U-shaped, three-line configuration using a Polyline element.

The first step to using a Polyline element is to import the Polyline package. The code for using polylines is different from the code for lines. For this reason, you need to import a separate package to work with polylines:

```
import javafx.scene.shape.Polyline;
```



**Figure 4-4** Three lines

With the package imported, once again place your cursor in the Scene's content attribute and type **Polyline**. Bring up the context menu to view the available attributes.

Notice that there are no definitions for startX and startY. Rather, a Polyline accepts one attribute, called "points." The points attribute requires an array of values.

### **NOTE**

An **array** is a collection of values that can be used and referenced as one value. In JavaFX, arrays appear between brackets.

### **TIP**

The content attribute accepts an array of values.

When you're passing an array of points to a Polyline element, it is understood that the first point will be the start point, and the last will be the end point. The Polyline code will instinctively draw your line between these points, stopping at all other points in your array. You can easily replicate the "U" formation of lines you created with Line using an array of points in Polyline.

An array of points that will match the points used in the Line example should look like this:

```
[10,10, 10,100, 100,100, 100,10]
```

Note that each value in the array is separated by a comma.

### **TIP**

To keep the array visually easy to understand, you can separate each x,y pair with a space. Whitespace within an array will be ignored by the compiler and can be added to make your array easy to read.

Run the following code. It should produce the same three-lined U shape as the previous example—with much less code:

```
package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Polyline;

/**
```

```
* @author JFDiMarzio
*/
Stage {
    title : "Basic Shapes"
    scene: Scene {
        width: 200
        height: 200
        content: [Polyline {
            points : [10,10, 10,100, 100,100, 100,10];
            strokeWidth : 15;
            stroke: Color.TOMATO
        }]
    }
}
```

Explore the remaining attributes of the Polyline element. You will find that many of the attributes match that of the line. This makes transitioning script from Line to Polyline quick and easy.

With this section complete, you have learned how to create simple lines and draw them to the screen. In the next section you will start to apply some of the knowledge you have picked up in the previous section and begin to create more complex shapes such as rectilinear and curvilinear objects.

## Rectangles

The JavaFX Rectangle element is used to refer to either a square or rectangle. It is used to draw any parallel-lined, four-sided shape. Given that the only difference between rectangles and squares is the length of a set of sides, it only makes sense that the same element can be used to draw either shape.

In the previous section you learned that a Polyline takes an array of points to create a multilined shape. The Rectangle refines this process by only requiring one point be specified. When you create a rectangle, you only need to specify the point for the upper-left corner of the shape. This point, when combined with a defined width and height, will give you a finished shape.

The code that is needed to create rectangles is contained within a separate package from that of the core JavaFX code. You will need to import the `javafx.scene.shape.Rectangle` package before you can create a rectangle. Therefore, insert the following import statements into your script:

```
import javafx.scene.shape.Rectangle;
import javafx.scene.paint.Color;
```

You are going to draw a rectangle that starts at the point x10, y10. It will be 100 pixels wide and 150 pixels high.

### NOTE

Do not get confused by the term *height*. The height of a rectangle is actually calculated from the start point, down (not the start point, up, as the name might imply). Therefore, a rectangle that starts at the point 1,1 and has a height of 100 will extend 100 pixels down from the point 1,1.

You will be specifying seven attributes for the Rectangle element:

- **x** The x coordinate for the upper-left corner of the rectangle
- **y** The y coordinate for the upper-left corner of the rectangle
- **width** The width of the rectangle
- **height** The height of the rectangle
- **fill** The color of the interior area of the rectangle
- **stroke** The attributes of the lines used to draw the rectangle
- **strokeWidth** The size of the line used to draw the rectangle

The first four attributes, shown next, are self-explanatory and are easy to determine the values of:

```
Rectangle{  
    x: 10;  
    y:10;  
    width: 100;  
    height: 150;  
}
```

The remaining three attributes may require a little explanation. All basic shapes, with the exceptions of lines and polylines, are filled in with Color.BLACK by default. For the purposes of the example you are building here, you only want to have the lines of the rectangle visible. Therefore, to have the rectangle not be filled with color and only expose the lines that create its border, you have to explicitly set the fill attribute to null.

Finally, the stroke and the strokeWidth attributes will be set just as they were in the previous section. That is, set the stroke to Color.BLACK and the strokeWidth to 5.

Keep in mind, the stroke and strokeWidth only refer to the lines of the rectangle, not the inner area of the rectangle.

Your finished Rectangle code should look like this:

```
Stage {  
    title : "Basic Shapes"  
    scene: Scene {  
        width: 200  
        height: 200  
        content: [Rectangle {  
            x : 10;  
            y : 10;  
            width : 100;  
            height : 150;  
            fill: null;  
            stroke: Color.BLACK;  
            strokeWidth : 5  
        }  
    ]  
}  
}
```

### TIP

To see what the rectangle looks like filled in, remove the fill attribute or set it to a specific color.

Notice that the line color is governed by the stroke attribute, whereas the interior color is governed by the fill attribute. This means that you can have a rectangle with a different line color and a different fill color. Here's an example:

```
Rectangle {  
    x : 10;  
    y : 10;  
    width : 100;  
    height : 150;  
    fill: Color.BLUE;  
    stroke: Color.RED;  
    strokeWidth : 5  
}
```

There are two more interesting attributes of the Rectangle element that are worth noting. If you want to round out the corners of your rectangle, you can use two attributes—arcWidth and arcHeight—to control the rounding. The attributes arcWidth

and arcHeight are used to define that amount of arc that is used when rounding the corners of the rectangle.

The arcWidth and arcHeight attributes are not available to Line or Polyline because this would basically turn the lines and polylines into arcs. Arcs are covered a bit later in this chapter.

Try the following code to round the corners of your rectangle:

```
Stage {
    title : "Basic Shapes"
    scene: Scene {
        width: 200
        height: 200
        content: [Rectangle {
            x : 10;
            y : 10;
            width : 100;
            height : 150;
            fill: null;
            stroke: Color.BLACK;
            strokeWidth : 5;
            arcWidth : 20;
            arcHeight : 20
        } ]
    }
}
```

Now that you have learned how to draw a rectangle on the screen, you can move on to a more complex shape—a polygon. The next section will discuss the process for drawing Polygons, or multisided shapes, to the screen using JavaFX script.

## Polygons

A polygon is to a rectangle what the polyline is to the line. Polygon elements are set up similarly to Polyline (discussed in a previous section of this chapter). Whereas Line and Rectangle accept a fixed number of coordinates to use a point, Polyline and Polygon accept an array of points.

Polygons will draw lines between all the points you specify in your points array, the same way Polyline does.

### **NOTE**

The same rules about color that apply to Rectangle elements also apply to Polygon elements. By default, all polygons are filled with black. Also, the line color and fill color are governed by the stroke and fill attributes, respectively.

The following sample code draws a simple, small octagon on the screen using the `Polygon` element:

```
package com.jfdimarzio.javafxforbeginners;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.shape.Polygon;

/**
 * @author jdimarz
 */

Stage {
    title : "Basic Shapes"
    scene: Scene {
        width: 400
        height: 400
        content: [Polygon {
            points: [
                90,80,
                190,80,
                240,130,
                240,220,
                190,270,
                90,270,
                40,220,
                40,130 ]
        }]
    }
}
```

Notice that you do not need to specify the final point of the polygon as being the same as the first point. JavaFX will automatically know that it should connect the last point in your array with the first point to complete the shape.

In the next section of this chapter you will learn how to create curvilinear shapes such as arcs, circles, and ellipses.

## Arcs

While on the surface an arc may seem like a simple shape, it can be very complicated for a computer to understand. People look at an arc and think, “It’s just a piece of a circle.” To a computer an arc is composed of complex radius and center points, angles, and circumference lengths.

An Arc element takes seven basic attributes. Given the complexity of drawing an arc as opposed to drawing a straight line, the attributes needed to complete an arc are unlike those you have seen so far in this chapter and therefore need some explanation. To begin, import the following packages:

```
import javafx.scene.shape.Arc;  
import javafx.scene.shape.ArcType;
```

The first two required attributes are centerX and centerY:

```
centerX: 125;  
centerY: 125;
```

Think of a circle for a moment. An arc would be a segment of that circle. JavaFX thinks of an arc as a circle that it only has to draw part of. Therefore, the centerX and centerY attributes represent the center point of the circle that the arc would form if it was completed.

The next two attributes are radiusX and radiusY:

```
radiusX: 50;  
radiusY: 50;
```

Every circle extends a certain distance from its center. This distance is the radius. The Arc element requires that you specify a radius. However, the naming of the attributes may be a little misleading. RadiusX and radiusY do not represent a point. Rather, they represent the radius in length along the x and y axes. Having two separate radial lengths lets you create oblong arcs.

### TIP

To make a circular arc, set the radiusX and radiusY attributes to the same value.

The next required attribute is startAngle:

```
startAngle: 45;
```

The startAngle represents an invisible line that extends from your center point to the actual start point of your arc. This line is at a given angle. Therefore, a startAngle of 45 mean that the arc will begin at a 45-degree angle to the center point.

The next attribute that is required to create an arc is the length:

```
length: 270;
```

The other length-related attributes you have worked with thus far, such as width and height, have been based on pixel length. The Arc attribute of length is a representation of the number of degrees that the arc travels from its start point. Therefore, if you have a length with a value of 270, your arc will extend 270 degrees from the start point, around the center point.

The next Arc attribute is type:

```
type: ArcType.OPEN;
```

The type attribute of the Arc element describes how the arc is drawn. Three different ArcType values can be assigned to this attribute:

- **ArcType.OPEN** Draws the arc as an open-ended curved line
- **ArcType.ROUND** Draws the arc while connecting the two end points back to the center point, much like a pie with a piece missing
- **ArcType.CHORD** Draws the arc and then connects the two end points to each other with a straight line

The last attribute you need to set is the fill attribute. The fill attribute is the same for an Arc element as it is for other shape elements. For the purposes of this example, set the fill attribute to null.

Optionally, you can set the stroke and strokeWidth attributes as you have done in the past sections. Run the code that follows to create an arc that looks like a pie with a missing piece:

### **NOTE**

You will need to import the `javafx.scene.paint.Color` package to fill your shape.

```
Stage {  
    title : "Basic Shapes"  
    scene: Scene {  
        width: 200  
        height: 200  
        content: [Arc {  
            centerX: 125;  
            centerY: 125;  
            radiusX: 50;  
            radiusY: 50;  
            startAngle: 45;  
            type: ArcType.CHORD;  
            fill: null;  
        }];  
    };  
};
```

```
        length: 270;
        type: ArcType.ROUND;
        fill: null;
        stroke: Color.BLACK;
        strokeWidth: 5
    }
]
}
}
```

Creating arcs, although it requires the use of several attributes that may seem foreign to you, becomes easier as you understand the purpose of each attribute.

In the next section you will create circles and ellipses.

## Circles and Ellipses

Creating a circle using JavaFX will seem much easier after you have created an arc. Whereas an Arc element needed seven attributes to govern angles, lengths, and other features, a Circle element only requires three. But before you can create a circle, you must import the appropriate package:

```
import javafx.scene.shape.Circle;
import javafx.scene.paint.Color;
```

The three attributes required to create a circle are centerX, centerY, and radius. These attributes function exactly like they do for an arc.

### NOTE

The reason a Circle element only has a single value for the radius and not a height and width pair for the radius, like Arc, is because a circle cannot be "oblong." Therefore, it will always have the same radial value along all axes.

Having a simple set of attributes makes circles easy to create and use in your applications. The following code creates a black outline of a circle:

```
Stage {
    title : "Basic Shapes"
    scene: Scene {
        width: 200
        height: 200
        content: [Circle {
            centerX: 100;
            centerY: 100;
            radius: 50;
        }]
    }
}
```

```
        fill: null;
        stroke: Color.BLACK;
        strokeWidth: 5
    }
]
}
}
```

You can use this same process to create an ellipse. The only difference between an ellipse and a circle is that because an ellipse is oblong, the Ellipse element takes an x, y radial pair rather than a set radial length. Here's the code:

```
/*
 * Chapter4.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 5/11/2010 - created
 *
 * Creating basic shapes
 */
package com.jfdimizaro.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.shape.Ellipse;
import javafx.scene.paint.Color;

Stage {
    title : "Basic Shapes"
    scene: Scene {
        width: 200
        height: 200
        content: [Ellipse {
            centerX: 50,
            centerY: 50,
            radiusX: 35,
            radiusY: 20,
            fill: null;
            stroke: Color.BLACK;
            strokeWidth: 5
        }]
    }
}
```

## Try This Create Multiple Shapes

Take some time out before the next chapter to exercise your new shape-building skills. Although it may seem like a fairly elementary task, you will have a great need to create simple shapes throughout your development career. From designing new buttons, to creating masks, you will always need to create simple shapes.

Try adding multiple shapes to the same Scene. Play with the dimensions and positions of the shapes to keep them from overlapping. Continue to work with them to manipulate how the shapes can touch or cover each other.

This exercise will be useful in future development by giving you a full grasp of how elements are placed in a Scene.

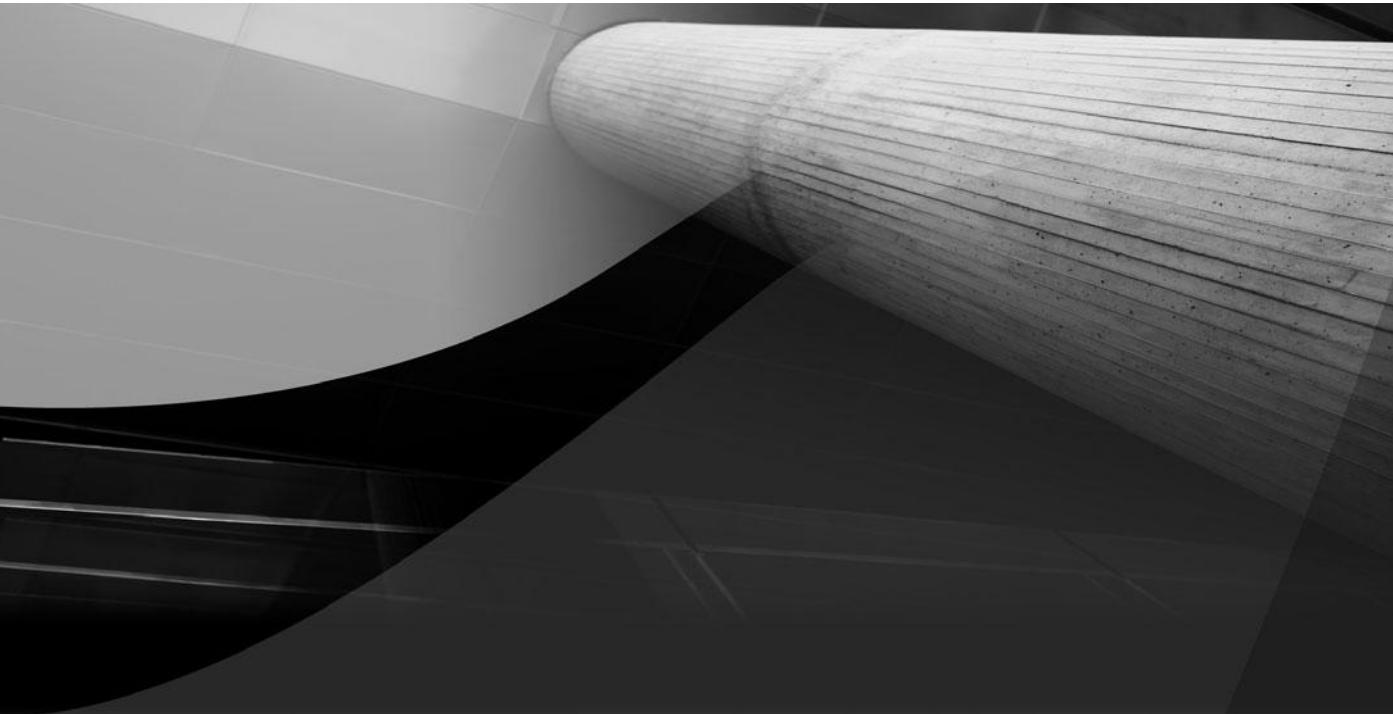
In this chapter, you learned how to create some basic shapes without using the Snippets menu. You created lines, polylines, rectangles, arcs, circles, and ellipses. You also learn about an invaluable tool: the context menu. These skills will help you tremendously throughout the remainder of this book.

In the next chapter, you will learn how to work with and create more colors than what was touched on in this chapter. You will also learn how to apply effects such as opacity and rotation.



## Chapter 4 Self Test

1. What four attributes are needed to draw a line?
2. How do you access the context menu?
3. What three delimiters can follow an attribute definition?
4. What attribute controls the thickness of the line used to draw a shape?
5. What package is needed to draw a polyline?
6. What type of value is assigned to the points attribute of a Polyline element?
7. True or false? The height attribute of the Rectangle element is the number of pixels from the start point to the top of the rectangle.
8. What is the default value for the fill attribute of a Rectangle element?
9. True or false? RadiusX and radiusY comprise the point where the radius extends to.
10. What attribute configures the radius of a circle?



# Chapter 5

Using Colors  
and Gradients

## Key Skills & Concepts

- Creating mixed colors
  - Applying colors to shapes
  - Using gradients
- 

In this chapter you explore more deeply the powerful coloring and gradient tools available to you in JavaFX. In Chapter 4 you learned how to apply some basic, pre-created colors to shapes. In this chapter you learn how to mix your own colors, and you apply these colors to shapes and use them in gradients.

## Using Color

The Color class is a very powerful class. In the last chapter you used a very small part of the Color class to fill a Line node with a solid color. You used just one or two of the 148 predefined colors of the Color class. This is a small sample of what the Color class is capable of.

The Color class can be used four different ways, each providing a capable method for creating the colors you need for any situation. You can invoke the Color class to use predefined colors, RGB values, HSB values, or web hex values.

In the next section you learn more about the predefined colors available to you. In the sections that follow you learn about the other color methods.

## Predefined Colors

The Color class, as contained in the `javafx.scene.paint.Color` package, has a number of predefined colors you can use instantly in the attributes of your nodes. In the previous chapter you used a predefined color (`Color.TOMATO`) to fill a line. In fact, all shapes use the `Color.BLACK` predefined color as the fill. The Color class provides the following predefined colors:

ALICEBLUE	ANTIQUEWHITE	AQUA
AQUAMARINE	AZURE	BEIGE
BISQUE	BLACK	BLANCHEDALMOND
BLUE	BLUEVIOLET	BROWN
BURLYWOOD	CADETBLUE	CHARTREUSE
CHOCOLATE	CORAL	CORNFLOWERBLUE
CORNSILK	CRIMSON	CYAN
DARKBLUE	DARKCYAN	DARKGOLDENROD
DARKGRAY	DARKGREEN	DARKGREY
DARKKHAKI	DARKMAGENTA	DARKOLIVEGREEN
DARKORANGE	DARKORCHID	DARKRED
DARKSALMON	DARKSEAGREEN	DARKSLATEBLUE
DARKSLATEGRAY	DARKSLATEGREY	DARKTURQUOISE
DARKVIOLET	DEPPINK	DEEPSKYBLUE
DIMGRAY	DIMGREY	DODGERBLUE
FIREBRICK	FLORALWHITE	FORESTGREEN
FUCHSIA	GAINSBORO	GHOSTWHITE
GOLD	GOLDENROD	GRAY
GREEN	GREENYELLOW	GREY
HONEYDEW	HOTPINK	INDIANRED
INDIGO	IVORY	KHAKI
LAVENDER	LAVENDERBLUSH	LAWNGREEN
LEMONCHIFFON	LIGHTBLUE	LIGHTCORAL
LIGHTCYAN	LIGHTGOLDENRODYELLOW	LIGHTGRAY
LIGHTGREEN	LIGHTGREY	LIGHTPINK
LIGHTSALMON	LIGHTSEAGREEN	LIGHTSKYBLUE
LIGHTSLATEGRAY	LIGHTSLATEGREY	LIGHTSTEELBLUE
LIGHTYELLOW	LIME	LIMEGREEN
LINEN	MAGENTA	MAROON
MEDIUMAQUAMARINE	MEDIUMBLUE	MEDIUMMORCHID
MEDIUMPURPLE	MEDIUMSEAGREEN	MEDIUMSLATEBLUE
MEDIUMSPRINGGREEN	MEDIUMTURQUOISE	MEDIUMVIOLETRED
MIDNIGHTBLUE	MINTCREAM	MISTYROSE

MOCCASIN	NAVAJOWHITE	NAVY
OLDLACE	OLIVE	OLIVEDRAB
ORANGE	ORANGERED	ORCHID
PALEGOLDENROD	PALEGREEN	PALETURQUOISE
PALEVIOLETRED	PAPAYAWHIP	PEACHPUFF
PERU	PINK	PLUM
POWDERBLUE	PURPLE	RED
ROSYBROWN	ROYALBLUE	SADDLEBROWN
SALMON	SANDYBROWN	SEAGREEN
SEASHELL	SIENNA	SILVER
SKYBLUE	SLATEBLUE	SLATEGRAY
SLATEGREY	SNOW	SPRINGGREEN
STEELBLUE	TAN	TEAL
THISTLE	TOMATO	TRANSPARENT
TURQUOISE	VIOLET	WHEAT
WHITE	WHITESMOKE	YELLOW
YELLOWGREEN		

A predefined color is accessed as a constant of the Color class. For example, in the Color class, the constant BLUE is associated with the values needed to create the blue color. Let's take a look at the following line of code (which is associated with a shape):

```
fill: Color.BLUE;
```

When you assign the fill attribute a value of Color.BLUE, the Color class passes along the values needed to create a blue color.

The Color class does offer a vast array of predefined colors to use in your scripts. But what if one of the predefined colors does not quite fit what you need? Fear not—the Color class has an even more powerful set of methods for rendering almost any color you could possibly need.

In the next section you learn about the methods available in the Color class for creating colors that are not predefined.

## Mixing Colors

If you have looked at the predefined colors but cannot find the exact one you need, you can always mix or specify your own color. The Color class would be a very limited tool if you could only use predefined colors without the ability to customize them.

For this reason, the Color class has three very useful methods for mixing your own colors:

```
Color.rgb();  
Color.hsb();  
Color.web();
```

You will learn about each of these methods and how to use them in the following sections.

## Color.rgb

The Color class allows you to use the RGB (Red, Green, Blue) value of the color you want to create. Most colors are mixed as a composite of differing amounts of red, green, and blue. The amounts of red, green, and blue used to create a color are defined using a value from 0 to 255 (0 being none of that color, 255 being a full amount of the specified color). The following code will fill the interior of a rectangle with purple:

```
Rectangle {  
    x : 10;  
    y : 10;  
    width : 150;  
    height : 150;  
    fill: Color.rgb(255,0,255);  
}
```

In this method call to the Color class, you specify an RGB value of 255 red, 0 green, and 255 blue. This produces a purple color. However, try the following code and see what you get:

```
Rectangle {  
    x : 10;  
    y : 10;  
    width : 150;  
    height : 150;  
    fill: Color{  
        red:1;  
        green:0;  
        blue:1}  
}
```

This code also produces a rectangle that is filled with purple. The default constructor for the Color class also accepts RGB values to create a color. So what is the difference? The end result is the same either way: You are left with purple as the color. The difference

is that the default constructor for the Color class accepts values for red, green, and blue as a float from 0 to 1. Keep in mind that the `rgb` method of the Color class accepts values from 0 to 255.

## Color.hsb

The Color class also has a method for HSB (Hue, Saturation, Brightness) colors. In the HSB color model, the color or hue is represented by a number from 0 to 360. This number corresponds to one of the 360 degrees of a color wheel.

The saturation and brightness attributes are represented by a number from 0 to 1. A value of 0 is no saturation and no brightness, whereas a value of 1 is full saturation and full brightness. To create the same purple color you just created with the `rgb` method, use the following code:

```
Rectangle {  
    x : 10;  
    y : 10;  
    width : 150;  
    height : 150;  
    fill: Color.hsb(300,1,1);  
}
```

## Color.web

Finally, the Color class can also use a web color hex value to create a color for you. The `web` method of the Color class accepts a standard hex value. Here's an example:

```
Rectangle {  
    x : 10;  
    y : 10;  
    width : 150;  
    height : 150;  
    fill: Color.web("#FF00FF");  
}
```

## The alpha Attribute

One last attribute of the Color class that you should take note of is the `alpha` attribute. Every method of the Color class has an optional `alpha` attribute. The `alpha` value controls the opacity of the color being created. The `alpha` is a value between 0 and 1, where 0 is transparent and 1 is opaque. The `alpha` value can be added to any of the methods of the Color class.

By setting the bit to a 0, you will achieve a full transparency.

```
Rectangle {  
    x : 10;  
    y : 10;  
    width : 150;  
    height : 150;  
    fill: Color.web("#FF00FF", 0);  
}
```

In contrast to the full transparency of 0, you can set the bit to 1 for a fully opaque color.

```
Rectangle {  
    x : 10;  
    y : 10;  
    width : 150;  
    height : 150;  
    fill: Color.web("#FF00FF", 1);  
}
```

Finally, setting the bit to .5 will give you a half transparency.

```
Rectangle {  
    x : 10;  
    y : 10;  
    width : 150;  
    height : 150;  
    fill: Color.web("#FF00FF", .5);  
}
```

In the next section you learn how to create and use gradients. Gradients provide a creative and eye-catching way to fill your shapes.

## Using Gradients

You can use two kinds of gradients to fill your shapes: `LinearGradients` and `RadialGradients`. `LinearGradients` are gradients that fill in a straight line from one side of a shape to the opposite side. `RadialGradients` start the gradient at one point, and the gradient radiates out from that point to fill the shape.

In the section that follows you learn how to apply a `LinearGradient` to a rectangle.

### LinearGradients

The `LinearGradient` class is contained in the `javafx.scene.paint.LinearGradient` package. Before you can work with `LinearGradients`, you must import this package:

```
import javafx.scene.paint.LinearGradient;
```

You need to learn about six parameters for the `LinearGradient` class before you can properly fill a shape with a gradient. The first is the `proportional` parameter, which accepts a Boolean value and determines how the remaining parameters are treated by the class. If the `proportional` value is set to true, the gradient fills the shape along a width that you specify. If the `proportional` value is set to false, you will need to explicitly set the beginning and end points for the gradient within your shape. This concept will make more sense as you move on.

### **NOTE**

If you do not specify a value for `proportional`, it will be set to true by default.

The next four parameters are `startX`, `startY`, `endX`, and `endY`. If `proportional` is set to true, these parameters should have a value between 0 and 1. On the x axis, 0 is the left side of your shape and 1 is the right side of your shape. On the y axis, 0 is the top of your shape and 1 is the bottom of your shape.

If `proportional` is set to true, the start and end parameters represent absolute points where the gradient should begin and end.

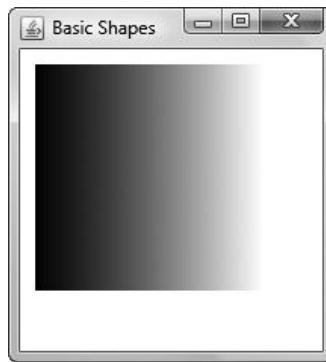
The final parameter is the `stops` parameter, which holds an array of colors that will be used in the gradient. This is a very versatile aspect of JavaFX: Gradients can be made with more than two colors. Each color you add to the gradient is composed of a `Color` class and an `offset` attribute. The `offset` is a number between 0 and 1 that determines where in the gradient the color is placed.

If your head is just about spinning right now, don't feel bad. `LinearGradients` will make much more sense when you see the code in action. Take a look at the following code.

### **NOTE**

Add the following import statement to your script `javafx.scene.paint.Stop`.

```
Rectangle {
    x : 10;
    y : 10;
    width : 150;
    height : 150;
    fill: LinearGradient {
        startX: 0.0;
        startY: 0.0;
        endX: 1.0;
        endY: 0.0;
        proportional: true;
```



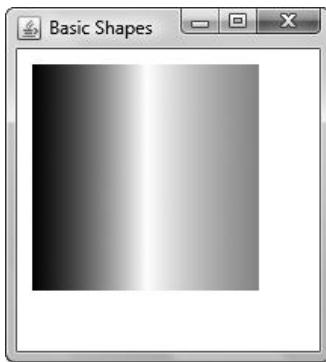
**Figure 5-1** A proportionally true, two-color LinearGradient

```
stops: [
    Stop {
        color: Color.BLACK,
        offset: 0.0
    },
    Stop {
        color: Color.WHITE,
        offset: 1
    }
]
}
```

The preceding code creates a LinearGradient as shown in Figure 5-1. This is a fairly standard two-color gradient.

The stops attribute accepts an array of colors. This means you can add as many colors as you want to the gradient. The following code creates the three-color LinearGradient shown in Figure 5-2:

```
Rectangle {
    x : 10;
    y : 10;
    width : 150;
    height : 150;
    fill: LinearGradient {
        startX: 0,
        startY: 0,
        endX: 1,
        endY: 0.0,
        proportional: true,
```



---

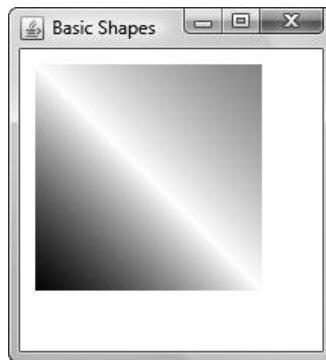
**Figure 5-2** A three-color LinearGradient

```
stops: [
    Stop {
        color: Color.BLACK,
        offset: 0.0
    },
    Stop {
        color: Color.WHITE,
        offset: .5
    }
    Stop {
        color: Color.TOMATO,
        offset: 1
    }
]
}
```

Notice how the value of the offset attribute changed to accommodate the three-color LinearGradient. When the gradient was composed of two colors, the offsets were 0 and 1. To place a third color exactly between the other two, an offset of .5 needed to be assigned. An offset higher than .5 would have placed the white closer to the right, whereas a value lower than .5 would have put it closer to the left.

By changing the start and end points along the y axis, you can tilt the gradient as shown in Figure 5-3.

In the next section you learn how to create and use a RadialGradient.



**Figure 5-3** A three-color skewed LinearGradient

## RadialGradients

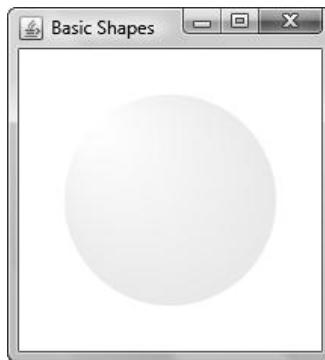
RadialGradients are gradients that, instead of emanating from one side, radiate out from a central point. Whereas LinearGradient work well with rectilinear shapes, RadialGradients work best on circles and ellipses. RadialGradients are in the `javafx.scene.paint.RadialGradient` package. You can import this package into your script using the following statement:

```
import javafx.scene.paint.RadialGradient;
import javafx.scene.shape.Circle;
```

Because a RadialGradient emanates out from a center point in a circular pattern, you must define a center point and radius. The center point and radius create a circle for the first color of the RadialGradient. The color will then fill the host shape while diffusing into the second color.

The RadialGradient class, like the LinearGradient class, also accepts an array of stops to produce gradients of more than two colors. The following code produces the gradient shown in Figure 5-4:

```
Circle {
    centerX: 100;
    centerY: 100;
    radius: 70;
    fill: RadialGradient {
        centerX: 5,
        centerY: 5,
        focusX: 0.1,
        focusY: 0.1,
        radius: 8,
```



---

**Figure 5-4** A RadialGradient in a circle

```
stops: [
    Stop {
        color: Color.WHITE,
        offset: 0.0
    },
    Stop {
        color: Color.BLACK,
        offset: 1.0
    }
]
}
```

## Try This Create a Custom Gradient

Using the skills you learned in Chapter 4, create a new Scene with multiple shapes on it. Once your shapes are in place, use the skills you learned in this chapter to add a different gradient to each one. Change the gradient by mixing different colors on each shape. This is a great way to practice multiple techniques and get instant results.

---

In this chapter you learned how to create colors, make gradients, and apply them to shapes. LinearGradients and RadialGradients provide an easy way to add visual interest to your shapes.

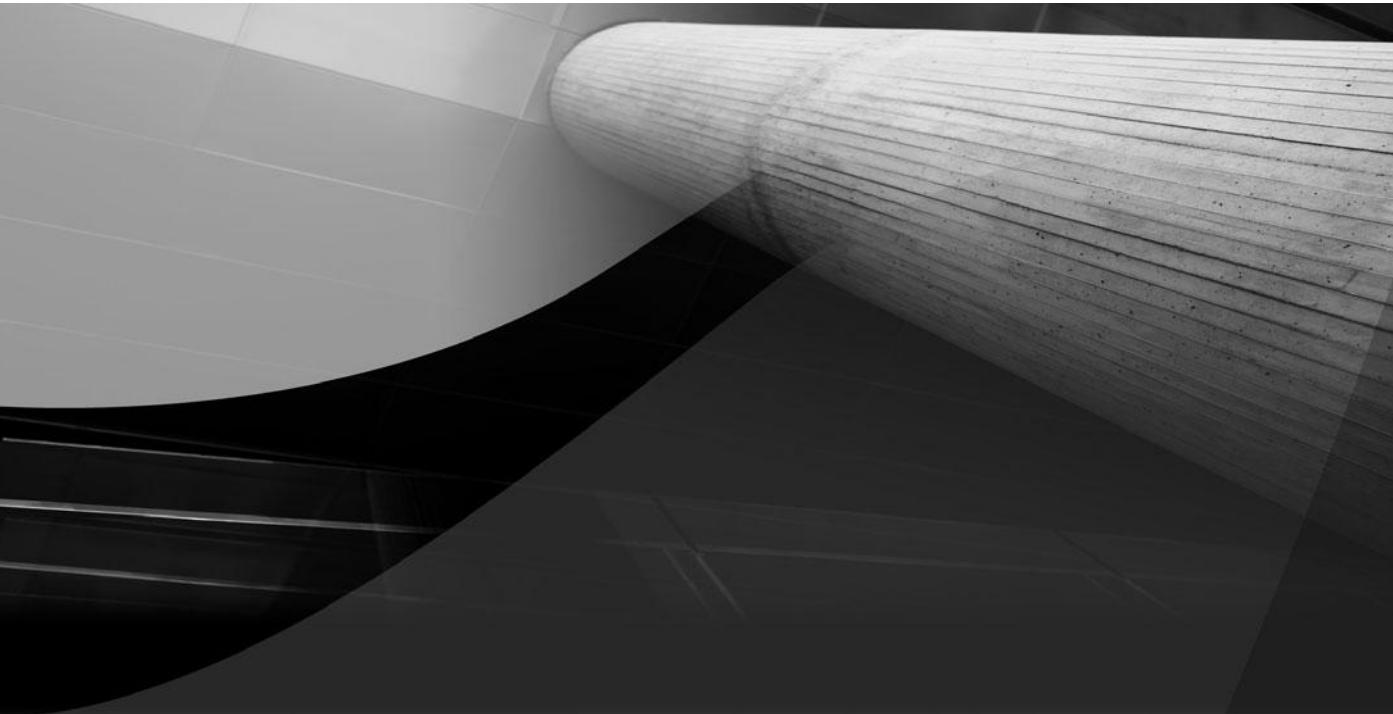
In the next chapter you will learn how to use images in your applications.



## Chapter 5 Self Test

1. How many predefined colors are available in the Color class?
2. What are the three methods available in the Color class for mixing colors?
3. True or false? RGB stands for refraction, gradient, and blur.
4. What is the acceptable value range for Hue?
5. In what package is the code for LinearGradients?
6. What is the default value for the proportional parameter?
7. What is the acceptable value for startX when proportional is set to true?
8. True or false? The stops parameter tells the gradient what point to stop on.
9. True or false? Gradients can be composed of more than two colors.
10. Which gradient is best for curvilinear shapes?

*This page intentionally left blank*



# Chapter 6

## Using Images

## Key Skills & Concepts

- Using an ImageView
  - Loading images
  - Loading an image placeholder
- 

**C**hances are that if you are making an application with JavaFX, you will need to have some form of interaction with images. Even if your application is not directly related to images or image working, you may need a splash screen, a background, or even an image for a control.

This chapter teaches you how to work with images and load them into JavaFX. By the end of this chapter you will be able to add an image file to your project and load it to the screen. The first step in this process is to learn about the ImageView node.

## The ImageView Node

Before you can display an image to the screen, you need to add an ImageView node to your application. All images are displayed using the ImageView node. Think of it as the film onto which your pictures are developed. The only purpose for the ImageView node is to display your images using the Image class.

The package for the ImageView is `javafx.scene.image.ImageView`. You must import this package before you can use the ImageView. Add a Stage and a Scene to your script, and then import the package for an ImageView, as follows:

```
/*
 * Chapter6.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 5/17/2010 - created
 *
 * Working with images
 */
package com.jfdimarzio.javafxforbeginners;
```

```
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.image.ImageView;

/**
 * @author JFDiMarzio
 */
Stage {
    title : "Images"
    scene: Scene {
        width: 200
        height: 200
        content: [
        ]
    }
}
```

You can insert an ImageView into the Scene's content using the following code:

```
content: [ImageView {
    image:
} ]
```

As it is now, the ImageView node will not do anything. Keep in mind that although the ImageView is a node by itself, it really doesn't do much without an image to display. Right now the script will not run; you need to add an image for the ImageView to display. In the next section you use the Image class to help the ImageView display something to the screen.

### **TIP**

The ImageView has attributes that affect the way in which an image is displayed. However, without an image to display, it does not make sense to discuss those attributes now. You will learn about these attributes after sending an image to the ImageView.

## The Image Class

The Image class is used to take in image files and format them for display. The Image class and the ImageView node work hand-in-hand to display your images to the screen.

The Image class is in the following package; it must be imported before you start working with the Image class:

```
import javafx.scene.image.Image;
```

The `Image` class can take in images from various sources. In this section you work with two of these sources: the Web and a local image file. First, let's pull an image from the Internet. The image you will display is <http://jfdimarzio.com/butterfly.png>.

Create an `Image` class and assign it to the `image` attribute of the `ImageView`. Notice that in the `Image` class you set the width and the height of the image. This is not necessarily the width and the height of the display (that is controlled by the width and height of the `Scene`). Rather, the width and the height of the `Image` class controls the size of the image that is passed to the `ImageView`.

The URL of the image—in this case, <http://jfdimarzio.com/butterfly.png>—is assigned as a value to the `url` parameter of the `Image` class. This tells the `Image` class where to look for a valid image to format. Set the `url` parameter as follows and then run your script:

```
scene: Scene {  
    width: 200  
    height: 200  
    content: [ImageView {  
        image: Image {  
            width: 200;  
            height: 200;  
            url: "http://jfdimarzio.com/butterfly.png"  
        }  
    }]  
}
```

After you run this script, your application should look like what's shown in Figure 6-1.

Because images loaded from the Web have a tendency to load slower, you may want to load the image in the background so that the user does not see a blank screen while the image is being displayed. This is easy to accomplish in JavaFX. In fact, you can use



---

**Figure 6-1** Using an image from the Web

a second image—a placeholder image—to display to your user while your main image is being loaded.

The following modification to the previous example loads the web image in the background. A second image is loaded in the placeholder and displayed to the user while the web image is loaded.

```
Stage {
    title : "Test"
    scene: Scene {
        width: 200
        height: 200
        content: [ImageView {
            image: Image {
                width: 200;
                height: 200;
                url: "http://jfdimarzio.com/butterfly.png"
                backgroundLoading: true;
                placeholder: Image{
                    url: "{__DIR__}images/waiting.png"
                }
            }
        } ]
    }
}
```

The placeholder attribute of the Image class takes its own Image class. This second Image class is used to display a temporary image when the main image is being downloaded. Notice the use of the backgroundLoading attribute to control whether or not the image is loaded in the background.

### NOTE

Previously you worked with the constant `{__PROFILE__}`. The constant `{__DIR__}` works the same way and is explained in more detail later in this chapter.

JavaFX does a good job of displaying images from a website, and the setup involved in this is fairly straightforward. The url parameter points directly to the image you want to display. However, chances are you are going to want to display images that you package into your application.

If you want to display an image that you have locally, the concept remains the same but the process is slightly different. You will want to think ahead as to how the image will be distributed with the application.

You can include an image in the package with your application. That image can then be called from an Image class and displayed using the ImageView. Distributing images in this way is more reliable than displaying them from the Internet. That is, there are more chances for something to go wrong if you are relying on an external website to host your images and you are relying on the Internet connection of that user to access the images.

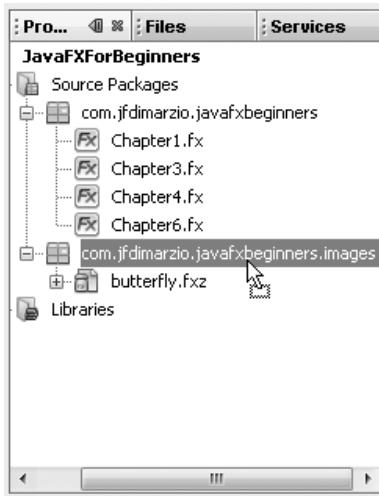
The first step in displaying a local image is to include that image in your package. Right-click your package and select New | Other. This will open the Create File dialog box. Select the Other category and then select a File Type of Folder. Click the Next button to continue and name your folder. Name the folder **images** and then click Finish. Now you have a folder within your package to keep your images.

### TIP

It is always recommended to create a separate folder for your images. This helps you keep your projects standardized and easy to manage.

Next, drag an image from your local drive into the images folder in the NetBeans IDE, as shown in Figure 6-2.

Now that the image is added to your project packages, you can reference it in your script. The key to referencing an image you have added directly to your package is a handy JavaFX constant called `{__DIR__}`.



---

**Figure 6-2** Adding an image to your package

The `{__DIR__}` constant represents the path to your package. The following string represents the contents of the `{__DIR__}` constant in my project (this constant will be different based on how you set up your project):

```
jar:file:/C:/Users/JFDiMarzio/Documents/NetBeansProjects/  
JavaFXForBeginners/dist/JavaFXForBeginners.jar!/com/jfdimizaro/  
javafxforbeginners/
```

You can reference your new image from your images folder by using the `{__DIR__}` constant to build your url value. The following code shows an `ImageView` displaying the `butterfly.png` file from the images folder:

```
scene: Scene {  
    width: 200  
    height: 200  
    content: [ImageView {  
        image: Image {  
            width: 200;  
            height: 200;  
            url: "{__DIR__}images/butterfly.png"  
        }  
    }]  
}
```

To this point you have learned how to display an image using the `Image` class and the `ImageView` node. Admittedly this is not the most exciting code in this book—it is actually very basic. However, there is one more way to display images in JavaFX that is unique and very powerful.

Layered images can be saved to a native JavaFX format known as an FXZ (Java FX Zip) file. JavaFX can load an FXZ file and display it as if it were any other image file. The advantage to using an FXZ file is that JavaFX can use the layer information to manipulate the image. To take full advantage of this final method of image display, you must learn about the JavaFX Production Suite.

## JavaFX Production Suite

The JavaFX Production Suite is a collection of tools that allow you to develop graphics for use within JavaFX scripts. More specifically, the core of the JavaFX Production Suite is a plug-in for Adobe Photoshop (CS3 and CS4) and Adobe Illustrator that exports your Adobe images to JavaFX FXZ files while preserving the layer information.

**NOTE**

The JavaFX Production Suite can be downloaded from the JavaFX website. It is a quick and rather painless installation.

Because the JavaFX Production Suite preserves the layer information of the images as used in Adobe Photoshop or Adobe Illustrator, you can access this valuable information within your JavaFX script. You can use this information to move, transform, and apply effects to each individual layer as if it were a separate image.

The key to ensuring that JavaFX can properly access the images is in how you set up your images in Adobe Photoshop or Adobe Illustrator.

**NOTE**

For the remainder of this chapter I will use Adobe Photoshop for the examples, but the same concepts apply to Adobe Illustrator.

Create an image that has layers in Adobe Photoshop. Figure 6-3 shows the butterfly image we have been using in this chapter. I have used the Quick Select Tool to create a new layer by cutting the butterfly from the background. The name you assign to each layer of your image is important. For the JavaFX Production Suite plug-in to preserve your layer naming, you must prefix each name with **jfx:**. For example, the image shown in Figure 6-3 consists of two layers: the background and the butterfly. To preserve these names within JavaFX, you must rename the layers, while inside Adobe Photoshop, to **jfx:background** and **jfx:butterfly**, respectively. The JavaFX Production Suite plug-in will know that any name prefixed with jfx: is to be preserved, and it will strip off the jfx: during the export process.

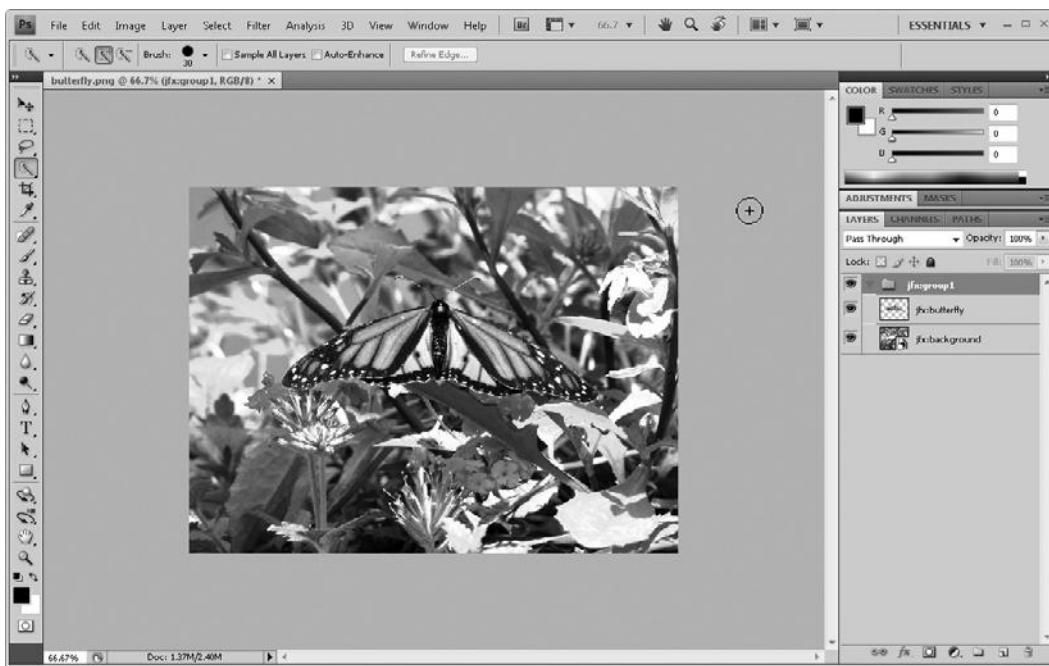
The import part of the process, as far as JavaFX is concerned, is what follows.

**TIP**

If you do not prefix the names of your layers with jfx:, the JavaFX Production Suite plug-in will generate names for you. This could be a problem if you want to retain control over how you access your images later.

Next, you must add your layers to a group. Again, this is done within Adobe Photoshop and can be seen in Figure 6-3. As with the layers, the group name must be prefixed with jfx:.

If you do not add your layers to a group, it will be much harder to access the individual layers. More code will be required to work with the layers after you export the image if you do not add them to a group while you are still in Adobe Photoshop.



**Figure 6-3** Butterfly image in Adobe Photoshop

**TIP**

I am not a Photoshop expert, but I have noticed that you will have a much easier time adding your layers to the group if you rename your layers before creating the group.

After your layers have been created and renamed, you can use the JavaFX Production Suite plug-in to export your image for use in JavaFX. Select File | Automate | Save for JavaFX... to save your file for use within your JavaFX script, as shown in Figure 6-4.

The JavaFX Production Suite plug-in will open an export options and preview window, as shown in Figure 6-5. You can use this window to preview what your exported image will look like if you want to. The image preview of the butterfly image is shown in Figure 6-6.

However, the important feature of this window is the option labeled Preserve ‘JFX:’ IDs Only. Make sure this option is selected to keep the layer and group names you applied while in Adobe Photoshop. Save your image when you are finished to complete the process.

The JavaFX Production Suite plug-in will create an FXZ file (for this example, the file is named butterfly.fxz). The butterfly.fxz file is just a compressed file that contains a content.fxd file and a number of images. You can open an FXZ file with a standard file decompression tool to see what is inside of it.

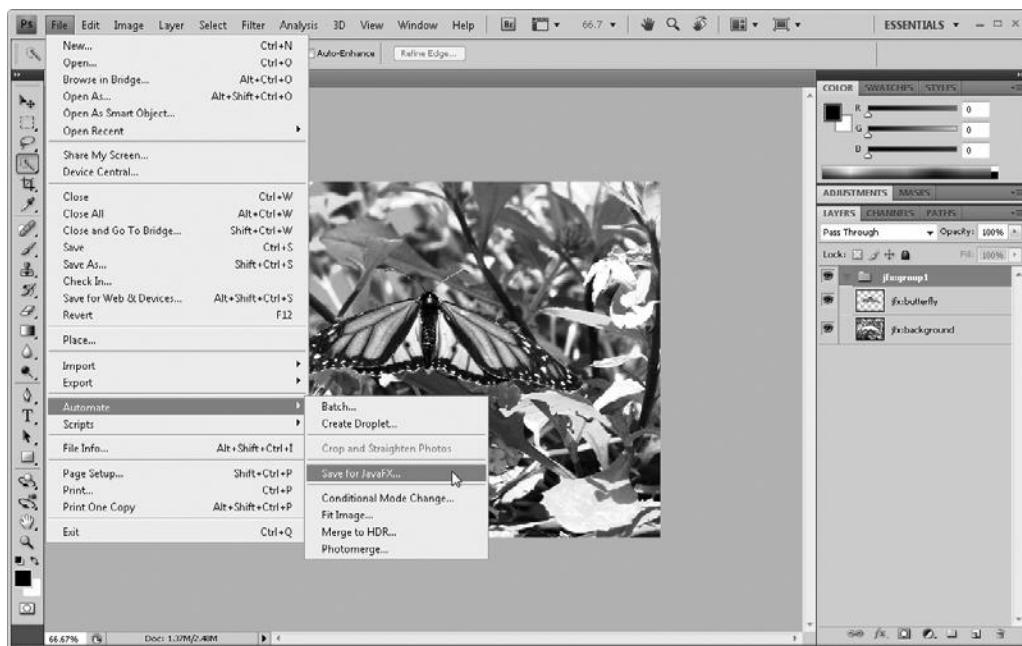


Figure 6-4 Saving your image for JavaFX

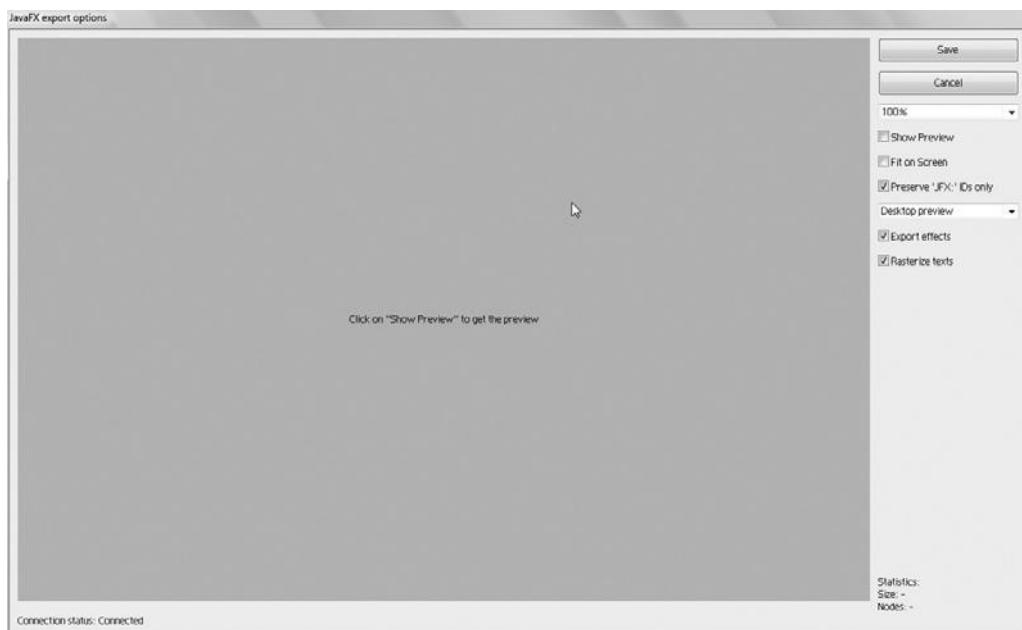


Figure 6-5 The JavaFX export options window



**Figure 6-6** Butterfly image preview in the JavaFX export options window

In the case of the butterfly.fxz file, it contains a content.fxd file and two images. The two images, butterfly.png and background.png, represent the two layers of the original image that were exported from Adobe Photoshop. These images were created by the JavaFX Production Suite plug-in. The images were named according to the layer names that were prefixed with the jfx: notation.

Although each image you export from Adobe Photoshop will likely have different layer images within its FXZ file, one element that will be common to all FXZ files is content.fxd, which is a definition file that specifies the relationship between the included image files. This is a JavaFX script that is imported into your project when you want to work with the image. The content.fxd file for the butterfly.fxz is shown here:

```
/*
 * Generated by JavaFX plugin for Adobe Photoshop.
 * Created on Sun Apr 11 17:45:24 2010
 */
//@version 1.0

Group {
    clip: Rectangle { x:0 y:0 width:800 height:600 }
    content: [
```

```
Group {
    id: "group1"
    content: [
        ImageView {
            id: "background"
            x: 0
            y: 0
            image: Image {
                url: "{__DIR__}background.png"
            }
        },
        ImageView {
            id: "butterfly"
            x: 151
            y: 181
            image: Image {
                url: "{__DIR__}butterfly.png"
            }
        },
    ]
},
```

}

Notice that the content.fxml file contains the code that creates a JavaFX Group. The Group contains a clip rectangle that defines the overall dimensions of the complete image and another Group that contains the individual images as ImageViews.

Now that you have a completed FXZ file (using the JavaFX Production Suite), you can use that file in your script.

## Using an FXZ File in JavaFX

Being able to use an FXZ file in your JavaFX script is a powerful and rewarding skill. It does not take much scripting to access an FXZ file. In fact, JavaFX provides a node just for the purpose of working with FXZ files. The FXDNode can be used to load images from an FXZ file.

The FXDNode is located in the javafx.fxml.FXDNode package. This package must be imported to load and work with FXZ files:

```
import javafx.fxml.FXDNode;
```

### NOTE

The following code example assumes the butterfly.fxz file has been included in the images folder within the current package.

The first step is to create the FXDNode and load the butterfly.fxz file. Create a var named butterflyGroup and type it as an FXDNode, as shown here:

```
var butterflyGroup : FXDNode = FXDNode{  
    url:"{__DIR__}images/butterfly.fxz"  
};
```

Now you can access the FXDNode by calling butterflyGroup. The url parameter points to the butterfly.fxz file in the images folder. Notice that the butterflyGroup var has been typed as an FXDNode using the : FXDNode notation. Although this is not necessary because JavaFX is not a strongly typed language, it is still a very good practice to get into.

Next, you can extract the butterfly image and move it to a different location against the background. The following code extracts the butterfly layer, moves it, and applies a rotation:

```
var butterfly = butterflyGroup.getNode("butterfly");  
butterfly.translateX = 50;  
butterfly.translateY = 50;  
butterfly.rotate = 45;
```

The getNode method of FXDNode is used to extract a layer from the loaded FXZ file. The getNode method takes the layer name that was applied to the layer before it was exported from Adobe Photoshop. In this case, you are extracting the butterfly layer; therefore, the name “butterfly” is passed to the getNode method.

The final step is to assign the FXDNode to the Scene’s content. The full script is as follows:

```
/*  
 * Chapter6.fx  
 *  
 * v1.0 - J. F. DiMarzio  
 *  
 * 5/17/2010 - created  
 *  
 * Working with images  
 *  
 */  
  
package com.jfdimizaro.javafxforbeginners;  
  
import javafx.stage.Stage;  
import javafx.scene.Scene;  
import javafx.fxml.FXDNode;  
  
var butterflyGroup : FXDNode = FXDNode{  
    url:"{__DIR__}images/butterfly.fxz"  
};
```

```
var butterfly = butterflyGroup.getNode("butterfly");
butterfly.translateX = 50;
butterfly.translateY = 50;
butterfly.rotate = 45;
/**
 * @author JFDiMarzio
 */
Stage {
    title : "FXZ Images"
    scene: Scene {
        content: [ butterflyGroup]
    }
}
```

Run this code using your Desktop profile and you will see the butterfly layer rotated and moved against the background, as shown in Figure 6-7.



---

**Figure 6-7** Butterfly layer rotated and moved

**Try This**

## Working with Different Image Types

Take some time before the next chapter to explore how JavaFX displays and works with different images and image types. Using the skills you learned in this chapter, try to display images of different types to JavaFX. Which images files will display? Which image files will not?

For an added level of research, try changing the sizes of the images. Take note of which image types allow you load up the largest images in the shortest amount of time.

This skill will help you in the future when you need to incorporate images of different types from different developers or sources into one application.

In this chapter you learned to import and display images to your JavaFX Scenes. This is an important skill to have in creating rich environments for your users. However, just displaying images alone will not create an exciting-enough environment to stop your users in their tracks.

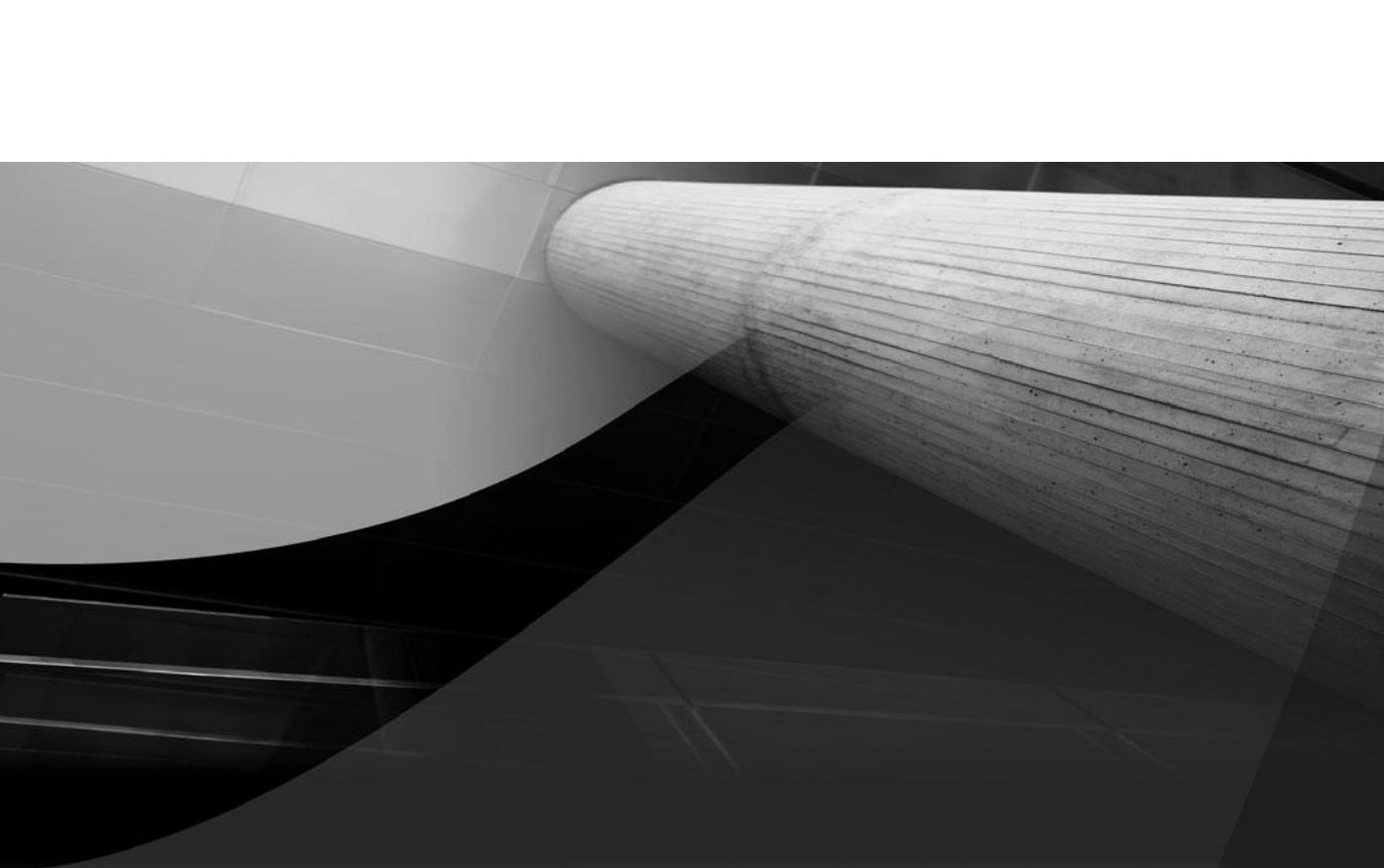
In Chapter 7 you learn how to apply effects to your images within JavaFX.



### Chapter 6 Self Test

1. What node is used to display images?
2. What class is used to write an image to the ImageView node?
3. True or false? An Image class can accept images from the Web.
4. What value does the {\_\_DIR\_\_} constant contain?
5. True or false? To have an image load in the background, use the BackgroundImage loader.
6. What is the name of the tool used to export images from Adobe Photoshop and Adobe Illustrator for JavaFX?
7. True or false? You must add jfx: to the beginning of each layer name to access those layers by name in your script.
8. What node is used to load images from an FXZ file?
9. True or false? The FXZ file is a compressed file that contains images and definitions.
10. What method is used to load an image layer?

*This page intentionally left blank*



# Chapter 7

## Applying Effects and Transformations

## Key Skills & Concepts

- Applying effects to shapes and images
  - Moving images in an application
  - Rotating images and shapes
- 

In Chapter 6 you learned how to write images to the screen using the FXDNode and ImageView nodes. In Chapter 4 you learned how to create different shapes and place them around your application. In this chapter you begin to apply effects and transitions to these images and shapes.

The first section of this chapter covers effects. JavaFX has a comprehensive list of effects you can apply to many objects within your scripts. These effects have a stunning impact on your applications and can be used to create almost any desired look or feel.

### **NOTE**

As of the 1.3 release of JavaFX, effects were not available to the Mobile profile. Therefore, you can only apply effects if your application is to be run in the Desktop or Browser profile.

To begin this chapter, create a new, empty JavaFX script (following the instructions covered earlier in the book). Name the script **Chapter7.fx** and save it to the project you have been working in throughout this book.

You are going to set up this script to display the same FXZ file you used in Chapter 6. This allows you to experiment with applying effects to a multilayered image. You will find that there are several ways to apply effects. For example, many effects can be applied to a group of images collectively or to individual images separately. Working with a two-image FXZ file will enable you to apply effects to an individual image as well as to the group.

Set up your script file as follows:

```
/*
 * Chapter7.fx
 *
 * v1.0 - J. F. DiMarzio
 */
```

```
* 5/20/2010 - created
*
* Applying Effects
*
*/
com.jfdimarzio.javafxforbeginners;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.fxml.FXDNode;
import javafx.scene.Group;
import javafx.scene.effect.*;
import javafx.scene.image.ImageView;

var imagePath : String = "{__DIR__}images/butterfly.fxz";
var butterflyImage : FXDNode = FXDNode{
    url: imagePath;
};

/**
 * @author JFDiMarzio
 */
Stage {
    title : "Effects"
    scene: Scene {
        width: 800
        height: 600
        content: [
            SetImages(butterflyImage)
        ]
    }
}
function SetImages(image : FXDNode) : FXDNode {
    var butterfly : ImageView;
    var background : ImageView;

    butterfly = (image.getNode("butterfly") as ImageView);
    background = (image.getNode("background") as ImageView);
    butterfly.translateX = -50;
    butterfly.translateY = -50;
    return image;
}
function ApplyEffects() : Effect{
    var effectToApply : Effect;
    effectToApply = null;
    return effectToApply;
}
```

This script may look a little confusing at first. However, it contains elements you have already learned about in earlier chapters.

The first section of the script contains two vars:

```
var imagePath : String = "{__DIR__}images/butterfly.fxz";
var butterflyImage : FDXNode = FDXNode{
    url: imagePath;
};
```

The first var holds the path to the FXZ file. The second var, butterflyImage, is the FDXNode of the ImageView group.

### TIP

Notice that the vars have been typed using the : <type> notation. Using this notation is not required for the script to compile, but it's a very good habit to get into.

Skipping over the middle of the script for a moment, let's look at the last section of the script, which contains two functions. The first function is as follows:

```
function SetImages(image : FDXNode) : FDXNode {
    var butterfly : ImageView;
    var background : ImageView;

    butterfly = (image.getNode("butterfly") as ImageView);
    background = (image.getNode("background") as ImageView);
    butterfly.translateX = -50;
    butterfly.translateY = -50;
    return image;
}
```

The first line of the function—the function definition—tells you that the function's name is SetImages and that it takes a single parameter and returns an FDXNode. The parameter that the function accepts is named image and is of the type FDXNode. This definition allows you to pass your butterflyImage var into the function, manipulate it, and then pass it back to the script. This is exactly the kind of function you need to experiment with some effects.

The second function, shown next, is where you put all the code for your effects:

```
function ApplyEffects() : Effect{
    var effectToApply : Effect;
    effectToApply = null;
    return effectToApply;
}
```

This function takes no parameters, but returns a value of Effect. Notice the var effectToApply, which is typed as an Effect. This var will be set to any effect you want to apply. That effect is then returned to the caller and rendered to the Scene. Throughout this chapter, you will use the ApplyEffects() function to apply different effects to the images.

Currently, the effectToApply is set to null, which applies no effect at all. As you progress through this chapter, you will replace the null with an effect to be applied.

**NOTE**

This design, where a common function is used to apply a specific effect to an image, may not be practical in a real working application. However, it lends itself very nicely to learning how effects work by separating the code.

The body of the function has much of the code you used in Chapter 6. The function takes the FXDNode that is passed into it and then creates three vars from it: one for the group, one for the butterfly, and one for the background. The butterfly image is moved -50 pixels along both the x and y axes and then returned to the script.

Finally, the middle of the script, where the Scene is, simply contains a call to the SetImages() function. When compiled, this script should produce an image like the one in Figure 7-1.



**Figure 7-1** The image before effects are applied

This script, in its current state, will be the base you use for the rest of this chapter. All the effects and transformations you learn about in this chapter are demonstrated using this script. The first section of this chapter covers effects, which is followed by a discussion of transformations.

## Effects

JavaFX is capable of rendering stunning and complex effects. Images and effects can be blended, blurred, and shadowed. Such effects can be very compelling and can be used to create applications that engage your users.

These effects can be applied to almost any node in JavaFX. All the standard effects available to you in JavaFX are in the `javafx.scene.effect` package. Import this package to begin working with JavaFX effects if you have not done so already:

```
import javafx.scene.effect.*;
```

### TIP

To this point, you have explicitly stated which items in each package you want to import. However, the `.*` notation tells JavaFX to import all the items within a particular package. Therefore, the statement `import javafx.scene.effect.*` will give you access to all the effects in the effect package.

The first effect you will use is Bloom.

## Bloom

The Bloom effect takes the areas of higher contrast in your image and makes them appear to glow by overrunning these areas of high contrast and bleeding them into the surrounding image. The amount of Bloom applied to the node is controlled by the threshold parameter. The threshold parameter accepts a value between 0 and 1, with 0 being no effect at all.

Use the `ApplyEffects()` and `SetImages()` functions in the `Chapter7` script to apply the Bloom effect to the background image of `butterfly.fxz`. Every `ImageView` node has an effect parameter. This parameter is set to the effect you want to apply to that `ImageView`.

### TIP

You are working with an `ImageView` in this chapter because you had just worked with it in Chapter 6. However, `ImageViews` are not the only node you can apply effects to. Almost every node contains an effect parameter that you can apply an effect to.

Add the following line to the SetImages() function to apply an effect to the background image:

```
background.effect = ApplyEffects();
```

Because the ApplyEffects() function has a return value of Effect, it can be passed directly to the effect parameter of the ImageView. If you want to apply the effect to the butterfly instead of the background, you can easily do so by using butterfly.effect rather than background.effect.

The SetImages() function should now look as follows:

```
function SetImages(image : FXDNode) : FXDNode {
    var butterfly : ImageView;
    var background : ImageView;

    butterfly = (image.getNode("butterfly") as ImageView);
    background = (image.getNode("background") as ImageView);
    butterfly.translateX = -50;
    butterfly.translateY = -50;
    background.effect = ApplyEffects();
    return image;
}
```

You can now edit the ApplyEffects() function to return the Bloom effect to the background. Add the following lines to the ApplyEffects() function:

```
effectToApply = Bloom{
    threshold: .5;
}
```

This sets effectToApply to the Bloom effect. The threshold is set to .5 to give you a good idea of what the Bloom effect can do. You can adjust the threshold to your liking to achieve the desired effect. Your ApplyEffects() function should now look like this:

```
function ApplyEffects() : Effect{
    var effectToApply : Effect;
    effectToApply = Bloom{
        threshold: .5;
    }
    return effectToApply;
}
```

The entire finished script is shown here:

```
/*
 * Chapter7.fx
 *
 * v1.0 - J. F. DiMarzio
```

```
*  
* 5/20/2010 - created  
*  
* Applying Effects  
*  
*/  
  
package com.jfdimarzio.javafxforbeginners;  
  
import javafx.stage.Stage;  
import javafx.scene.Scene;  
import javafx.fxml.FXDNode;  
import javafx.scene.Group;  
import javafx.scene.image.ImageView;  
import javafx.scene.effect.*;  
  
var imagePath : String = "{__DIR__}images/butterfly.fxz";  
var butterflyImage : FXDNode = FXDNode{  
    url: imagePath;  
};  
Stage {  
    title : "Effects"  
    scene: Scene {  
        width: 800  
        height: 600  
        content: [  
            SetImages(butterflyImage)  
        ]  
    }  
}  
function SetImages(image : FXDNode) : FXDNode {  
    var butterfly : ImageView;  
    var background : ImageView;  
  
    butterfly = (image.getNode("butterfly") as ImageView);  
    background = (image.getNode("background") as ImageView);  
    butterfly.translateX = -50;  
    butterfly.translateY = -50;  
    background.effect = ApplyEffects();  
    return image;  
}  
function ApplyEffects() : Effect{  
    var effectToApply : Effect;  
    effectToApply = Bloom{  
        threshold: .5;  
    }  
    return effectToApply;  
}
```

Compile your script and run it to produce an image with the Bloom effect applied, as shown in Figure 7-2.

Next, you learn about the ColorAdjust effect.

## ColorAdjust

As the name suggests, the ColorAdjust effect allows you to adjust the color of your node. In much the same way that you adjust the picture on your television, JavaFX lets you adjust your images. ColorAdjust contains parameters that allow you to adjust the contrast, brightness, hue, and saturation.

All the parameters for the ColorAdjust effect, with the exception of input, accept a numeric value. You can assign a value between –1 and 1 to each of the following: contrast, brightness, hue, and saturation. However, you do not necessarily need to assign a value



**Figure 7-2** Background with the Bloom effect

to all the parameters. For instance, if you just want to adjust the contrast of an image, you only have to specify a value for the contrast parameter, and JavaFX will automatically assign a 0 to the others. (Note that JavaFX will not auto-assign a 0 to contrast because it has a default value of 1.)

For this example, you will assign .5 to each parameter and then assign the effect to the background. Add the following code to the `ApplyEffects()` function:

```
effectToApply = ColorAdjust{  
    contrast : .5;  
    brightness : .5;  
    hue : .5;  
    saturation : .5;  
}
```

The structure of the `ColorAdjust` effect should look familiar if you followed along with the previous example. Effects, as a whole, are not very complicated to apply once you have done it a couple times. Your complete `ApplyEffects()` function should look like this:

```
function ApplyEffects() : Effect{  
    var effectToApply : Effect;  
    effectToApply = ColorAdjust{  
        hue : .5;  
        saturation : .5;  
        brightness : .5;  
        contrast : .5;  
    }  
    return effectToApply;  
}
```

Compile your script and run it with the Desktop profile. You will see an image like the one shown in Figure 7-3.

Notice that the background colors have been adjusted and appear almost animated. However, the butterfly has not been affected. Experiment with assigning different values to each parameter—and to the butterfly and background independently.

The next effect you learn about is `GaussianBlur`.

## GaussianBlur

`GaussianBlur` provides a very smooth blurring effect to a node. Behind the scenes, the effect works on something called the *Gaussian algorithm*. This algorithm works in a circular pattern from each pixel to smooth the appearance. Because this algorithm works in a circular pattern, the parameter you need to work with is the radius parameter, which controls how far out from each pixel the Gaussian algorithm is applied.



**Figure 7-3** Background modified using ColorAdjust

You can specify a value from 0 to 63 for the radius of the GaussianBlur. A value of 0 would be little to no blur in the original image, whereas a value of 63 would be an extreme blur. The following code implements a GaussianBlur with a radius of 10:

```
effectToApply = GaussianBlur{
    radius: 10;
}
```

Apply the GaussianBlur to your ApplyEffects() function as follows:

```
function ApplyEffects() : Effect{
    var effectToApply : Effect;
    effectToApply = GaussianBlur{
        radius: 10;
    }
    return effectToApply;
}
```

Run your completed script, which blurs the background with a radius of 10. The result is shown in Figure 7-4.

Notice that the background is slightly blurred, but still recognizable. Now change the radius to 60 and recompile the script. The results are shown in Figure 7-5.

GaussianBlur is particularly effective when you are trying to resize images. Occasionally an image can become distorted when it is resized, especially if the image is quite sharp before it is resized. Applying a very light GaussianBlur before resizing the image (in systems that do not do so automatically) can make the resized image less distorted.

You learn about the Glow effect next.



---

**Figure 7-4** Background with a GaussianBlur radius of 10



**Figure 7-5** Background with a GaussianBlur radius of 60

## Glow

The Glow effect, as the name suggests, makes your node appear to glow. The amount of glow applied to the node is controlled by the level parameter. The level of glow you can apply to a node ranges from 0 to 1. Assigning no level for the Glow effect will cause JavaFX to use a default of .3.

Earlier in this chapter you learned about the Bloom effect, which applied a Glow-like effect to your node. The difference between Bloom and Glow is in the way the glow is applied. Whereas Bloom only applies a glow to the higher contrast parts of an image, Glow works on the entire image.

The following code shows your `ApplyEffects()` function with a `Glow` level of `.5`:

```
function ApplyEffects() : Effect{
    var effectToApply : Effect;
    effectToApply = Glow{
        level: 1
    }
    return effectToApply;
}
```

Replace the `ApplyEffects()` function in your current script with this one. Your full script should now look like this:

```
/*
 * Chapter7.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 5/20/2010 - created
 *
 * Applying Effects
 */

```

```
package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.fxml.FXDNode;
import javafx.scene.Group;
import javafx.scene.image.ImageView;
import javafx.scene.effect.*;
```

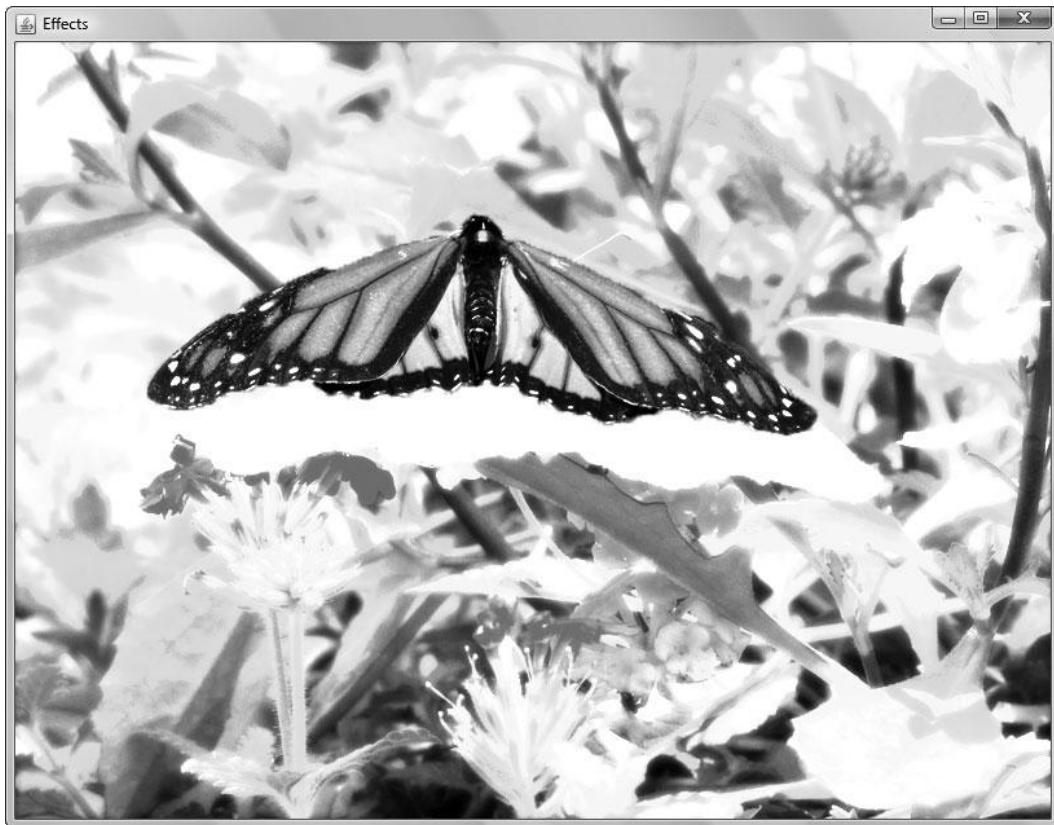
```
var imagePath : String = "{__DIR__}images/butterfly.fxz";
var butterflyImage : FXDNode = FXDNode{
    url: imagePath;
};

Stage {
    title : "Effects"
    scene: Scene {
        width: 800
        height: 600
        content: [
            SetImages(butterflyImage)
        ]
    }
}
function SetImages(image : FXDNode) : FXDNode {
    var butterfly : ImageView;
    var background : ImageView;
```

```
        butterfly = (image.getNode("butterfly") as ImageView);
        background = (image.getNode("background") as ImageView);
        butterfly.translateX = -50;
        butterfly.translateY = -50;
        background.effect = ApplyEffects();
        return image;
    }
    function ApplyEffects() : Effect{
        var effectToApply : Effect;
        effectToApply = Glow{
            level: 1
        }
        return effectToApply;
}
```

Compile and run your script. Your background image should glow like the one shown in Figure 7-6.

The next effect you learn about is the DropShadow effect.



**Figure 7-6** Background with a Glow of .5

## DropShadow

The DropShadow effect creates a shadow under your node by replicating the node in a shadow color and offsetting the “shadow” image by a specific amount under your source node. Quite a few parameters are needed to configure DropShadow:

- **radius** Used like the radius parameter for GaussianBlur.
- **height/width** Can be used instead of radius; has the same effect.
- **spread** The opacity of the shadow. A value of 0 creates a very light, scattered shadow, whereas 1 produces a dark, sharp shadow.
- **blurType** The algorithm used to create the shadow. This can be set to Gaussian, ONE\_, TWO\_, or THREE\_PASS\_BOX.
- **color** The color of the shadow; defaults to BLACK.

Modify your ApplyEffects() function to create a DropShadow effect, as follows:

```
function ApplyEffects() : Effect{  
    var effectToApply : Effect;  
    effectToApply = DropShadow{  
        radius : 10;  
        offsetX: 10;  
        offsetY: 10;  
        spread: .2;  
        blurType : BlurType.THREE_PASS_BOX;  
    }  
    return effectToApply;  
}
```

Rather than apply this effect to the background as you have been doing, try applying it to the butterfly. Take a look at the full script to see how this effect is applied to the butterfly:

```
/*  
 * Chapter7.fx  
 *  
 * v1.0 - J. F. DiMarzio  
 *  
 * 5/20/2010 - created  
 *  
 * Applying Effects  
 *  
 */
```

```
package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.fxml.FXDNode;
import javafx.scene.Group;
import javafx.scene.image.ImageView;
import javafx.scene.effect.*;

var imagePath : String = "{__DIR__}images/butterfly.fxz";
var butterflyImage : FXDNode = FXDNode{
    url: imagePath;
};

Stage {
    title : "Effects"
    scene: Scene {
        width: 800
        height: 600
        content: [
            SetImages(butterflyImage)
        ]
    }
}

function SetImages(image : FXDNode) : FXDNode {
    var butterfly : ImageView;
    var butterflyShadow : ImageView;
    var background : ImageView;

    butterfly = (image.getNode("butterfly") as ImageView);
    background = (image.getNode("background") as ImageView);
    butterfly.translateX = -50;
    butterfly.translateY = -50;
    butterfly.effect = ApplyEffects();
    return image;
}
function ApplyEffects() : Effect{
    var effectToApply : Effect;
    effectToApply = DropShadow{
        radius : 10;
        offsetX: 10;
        offsetY: 10;
        spread: .2;
        blurType : BlurType.THREE_PASS_BOX;
    }
    return effectToApply;
}
```

This script produces a shadow under the butterfly like the one shown in Figure 7-7.

The DropShadow effect works by creating a blurred copy of your original image. The copy is placed under the original image and then offset so you can see it. You can use the Shadow effect if you want to control this process yourself.

The Shadow effect creates a blurred, colored image based on your original. However, it does not re-add an unaltered copy of your image to the scene. You are literally just left with the shadow of your image. You will have to manually add another instance of your image to the scene to complete the effect.

Using the Shadow effect over the DropShadow effect has its advantages, though. For example, it can come in quite handy if you want to project the shadow onto a location detached from the original image.

The next effect you learn about is the InvertMask effect.



**Figure 7-7** Butterfly with DropShadow

## InvertMask

InvertMask is a simple effect that inverts the opacity of your node. That is, any part of your node that is transparent becomes opaque, and any part that is opaque becomes transparent.

Alter your `ApplyEffects()` function as follows to apply the InvertMask effect to the butterfly:

```
function ApplyEffects() : Effect{  
    var effectToApply : Effect;  
    effectToApply = InvertMask{  
    }  
    return effectToApply;  
}
```

Compile this into your script and run it. Notice in Figure 7-8 that you can now see the bounding box that was around the butterfly image.

You learn about the Lighting effect next.



**Figure 7-8** The InvertMask effect used on the butterfly image

## Lighting

The Lighting effect is by far the most complex effect offered in JavaFX. Lighting can be used to add a dimension of realism to an otherwise flat object. Although Lighting is a complex effect to set up, it is very rewarding if used correctly.

The main parameter of the Lighting effect is Light, which represents the type of light from the `javafx.scene.effect.light` package. Three different types of light can be used and assigned to the Lighting effect:

- DistantLight
- PointLight
- SpotLight

### **NOTE**

The relationship between Lighting and Light is that Lighting defines how Light is used by the effect.

Each of these Light types has its own set of parameters that controls the specific type of light.

### DistantLight

DistantLight takes three parameters that configure and control the light. Here's the value map of DistantLight:

```
DistantLight{  
    azimuth : <angle of the light in degrees>  
    elevation : <elevation of the light to the object in degrees>  
    color : <Color of light>  
};
```

Update your `ApplyEffects()` function to implement the Lighting effect using DistantLight. Set the azimuth of the light to 45, the elevation to 45, and the color to RED, as shown next. Do not forget to import the `javafx.scene.paint.Color.*` package to manipulate the color of the light.

```
function ApplyEffects() : Effect{  
    var effectToApply : Effect;  
    effectToApply = Lighting{  
        light : DistantLight{  
            azimuth : 45;  
            elevation : 45;  
            color : RED;
```

```
    };  
}  
return effectToApply;  
}
```

Compile your script and run it to apply a red-colored distant lighting effect to the butterfly. The result is shown in Figure 7-9.

### PointLight

The value map of PointLight is as follows:

```
PointLight{  
    x: <x position of light source in 3D space> ;  
    y: <y position of light source in 3D space>;  
    z: <z position of light source in 3D space>;  
    color : <color of light>;  
};
```



**Figure 7-9** Butterfly with red DistantLight applied

Just as you did with DistantLight, edit your ApplyEffects() function to use a PointLight at position x150, y50, z50, as shown here:

```
function ApplyEffects() : Effect{
    var effectToApply : Effect;
    effectToApply = Lighting{
        light : PointLight{
            x: 150;
            y: 50;
            z: 50;
            color : YELLOW;
        };
    }
    return effectToApply;
}
```

Your image should look like Figure 7-10 after you compile and run your script.



**Figure 7-10** Butterfly with a yellow PointLight

## SpotLight

SpotLight takes the same parameters as PointLight, but also adds a few others to guide where the light is pointing. These parameters are pointsAtX, pointsAtY, and pointsAtZ.

The following ApplyEffects() function will apply a SpotLight lighting effect to the butterfly:

```
function ApplyEffects() : Effect{
    var effectToApply : Effect;
    effectToApply = Lighting{
        light : SpotLight{
            x: 150;
            y: 50;
            z: 50;
            pointsAtX: 400;
            pointsAtY: 50;
            pointsAtZ: 0;
            color : WHITE;
        };
    }
    return effectToApply;
}
```

Compile your script; the image will look like the one shown in Figure 7-11.

Next, you learn about the SepiaTone effect.

## SepiaTone

The SepiaTone effect is designed to emulate the look of older black-and-white film. Early film was tinted with Sepia to add color to the prints. This process can be emulated with the SepiaTone effect, which takes a level parameter to adjust the amount of effect applied to the node. The level can be a value between 0 and 1.

Take a look at the following full script; notice that the effect has been applied to the group image rather than just the butterfly:

```
/*
 * Chapter7.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 5/20/2010 - created
 *
 * Applying Effects
 */

```



**Figure 7-11** Butterfly with SpotLight

```
package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.fxml.FXDNNode;
import javafx.scene.Group;
import javafx.scene.image.ImageView;
import javafx.scene.effect.*;

var imagePath : String = "{__DIR__}images/butterfly.fxz";
var butterflyImage : FXDNNode = FXDNNode{
    url: imagePath;
};
Stage {
    title : "Effects"
    scene: Scene {
```

```
width: 800
height: 600
content: [
    SetImages(butterflyImage)
]
}
}

function SetImages(image : FXDNode) : FXDNode {
    var butterfly : ImageView;
    var butterflyShadow : ImageView;
    var background : ImageView;
    butterfly = (image.getNode("butterfly") as ImageView);
    background = (image.getNode("background") as ImageView);
    butterfly.translateX = -50;
    butterfly.translateY = -50;
    groupImage.effect = ApplyEffects();
    return image;
}

function ApplyEffects() : Effect{
    var effectToApply : Effect;
    effectToApply = SepiaTone{
        level: 1;
    }
    return effectToApply;
}
```

Figure 7-12 shows the results of this script.

In the next section of this chapter, you learn about transformations and how they differ from effects.

## Transformations

A transformation does not change the node the way an effect does. The major purpose of a transformation is to move or shift the node along an axis. You have already been working with transformations in this chapter, even if you were unaware of it.

The three different kinds of transformations are xy transformations, rotations, and transformations of perspective. These transformation types are detailed next.

### XY Transformations

In this chapter you have been working with a script that contains two ImageView nodes: background and butterfly. Take a look at the following SetImage() function. In this function,



**Figure 7-12** SepiaTone applied to the group image

you move the butterfly ImageView –50 pixels on the x axis and –50 pixels on the y axis using transformations:

```
function SetImages(image : FXDNode) : FXDNode {
    var butterfly : ImageView;
    var butterflyShadow : ImageView;
    var background : ImageView;

    butterfly = (image.getNode("butterfly") as ImageView);
    background = (image.getNode("background") as ImageView);
    butterfly.translateX = -50;
    butterfly.translateY = -50;
    groupImage.effect = ApplyEffects();
    return image;
}
```

The ImageView node has attributes named `translateX` and `translateY`. These attributes are used to move the ImageView around the Scene.

## Rotation

Rotating an ImageView is just as easy as moving it along an axis. The ImageView node has an attribute named `rotate`. Simply set the `rotate` attribute of the ImageView to the number of degrees you want to rotate the image. For example, take a look at the following `SetImage()` function, which rotates the butterfly image 45 degrees. The result is shown in Figure 7-13.



**Figure 7-13** Butterfly rotated 45 degrees

**NOTE**

Remove the Sepia effect from `ApplyEffects()` to return the butterfly to normal before you compile this example.

```
function SetImages(image : FXDNode) : FXDNode {
    var butterfly : ImageView;
    var butterflyShadow : ImageView;
    var background : ImageView;

    butterfly = (image.getNode("butterfly") as ImageView);
    background = (image.getNode("background") as ImageView);
    butterfly.rotate = 45;
    butterfly.effect = ApplyEffects();
    return image;
}
```

Next you learn about `PerspectiveTransform`—a member of the `Effects` package that transforms the perspective of your node.

## PerspectiveTransform

`PerspectiveTransform` alters the perspective of a node by giving you control over the x and y coordinates of each corner of the node.

`PerspectiveTransform` takes eight parameters for the x and y axes of the upper-left, upper-right, lower-left, and lower-right corners. Take a look at the following code for the `ApplyEffects()` function:

```
function ApplyEffects() : Effect{
    var effectToApply : Effect;
    effectToApply = PerspectiveTransform{
        ulx : 100;
        uly : 100;
        urx : 400;
        ury : 100;
        lrx : 400;
        lry : 550;
        llx : 100;
        lly : 350;
    };
    return effectToApply;
}
```

In this example, all the coordinates for each corner have been set. It takes a little practice to get the desired effect when using the `PerspectiveTransform`, and you will want to experiment with the different coordinates. The preceding code produces the result shown in Figure 7-14.



**Figure 7-14** Butterfly with PerspectiveTransform

## Try This Combining Multiple Effects

The saying “You can’t have too much of a good thing” is especially true of JavaFX effects. It is rare when working with an image that you will have to apply just one effect. Often, you will need to apply multiple effects to an image to get the desired look.

Using the skills you acquired in this chapter, create a new project with an image in it. Apply multiple effects to the image at the same time, and adjust the properties of these effects to create new and exciting images.

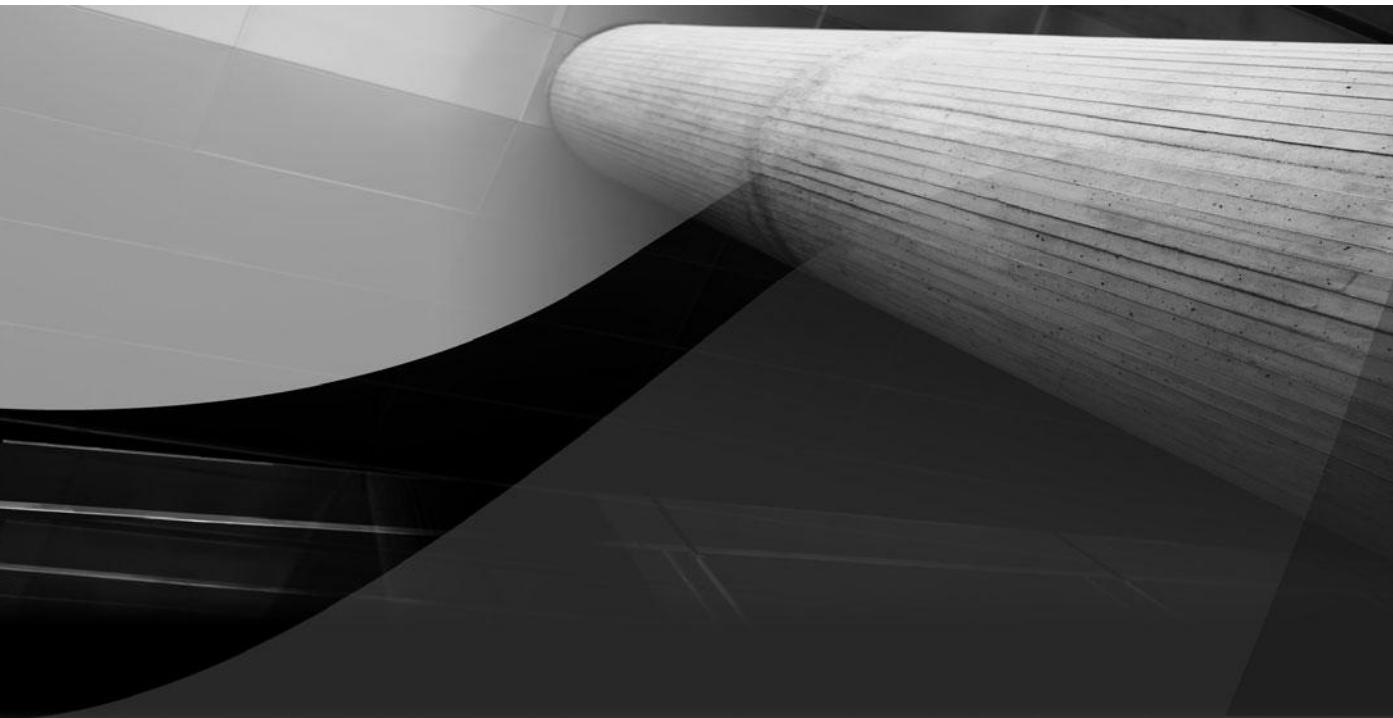
This concludes the chapter on effects and transformations. In the next chapter you will begin to tackle some basic animation.



## Chapter 7 Self Test

1. How do you assign a type to a var?
2. What effect adjusts only the higher contrast areas of your node to make them glow?
3. True or false? All the parameters of ColorAdjust default to 0 if they are not specified.
4. What parameter needs to be specified to create a GaussianBlur effect?
5. What is the difference between Glow and Bloom?
6. True or false? You do not need to specify both a radius and a height/width for a DropShadow effect.
7. Which effect takes all the opaque areas of your image and makes them transparent?
8. What are the three different lights that can be used in the Lighting effect?
9. What does the following code do?

```
butterfly.rotate = 45;
```
10. How many parameters need to be set to create a PerspectiveTransform effect?



# Chapter 8

## Basic Animation

## Key Skills & Concepts

- Using Timelines
  - Creating paths
  - Using keyframes
- 

This chapter introduces you to the world of basic JavaFX animation. Whether you want to create animated text and fly-ins or gain knowledge for creating games, basic animation is the place to begin.

You will need to master three basic topics when tackling basic JavaFX animation:

- Timelines
- Keyframes
- Paths

To begin this chapter, open NetBeans to the JavaFXForBeginners project you have been working on throughout this book. Create a new, empty JavaFX script file named **Chapter8.fx**. Based on previous chapters, the contents of this file should look as follows:

```
/*
 * Chapter8.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 5/27/2010 - created
 *
 * Basic Animation
 *
 */
package com.jfdimarzio.javafxforbeginners;

/**
 * @author JFDiMarzio
 */

// place your code here
```

The first section of this chapter covers Timelines.

## Timelines

All animation, whether it is “traditional” hand-drawn animation or computer-based animation, is controlled by timing. What actions occur and when, the length of time it takes to walk from one side of a room to another, and syncing the dialogue to a character’s mouth movements are all actions that are controlled by some sense of timing. The timing of the animation dictates when each action begins, when it ends, and how long it lasts.

Timing is critical to smooth animation. If there is too much time between each frame of animation, it will look slow and jerky to the user. If there is too little time between each frame, the animation will be too fast. This is why timing is so critical.

In JavaFX, the timing of animation is controlled by a Timeline. A Timeline takes in keyframes and outputs values that help you control your animation on the screen. The package that holds JavaFX Timelines is `javafx.animation.Timeline`.

The purpose of a Timeline is to break down frames of animation into “stops,” by time. This means that if you tell a Timeline where you want an object to be one second from now, and then five seconds from now, the Timeline will produce for you a value that will help your object move. The Timeline takes on the responsibility of producing a smooth increment of values that represent the movement of your object over the time you specify in your keyframes. This may sound a bit confusing now, but it will make a lot more sense when you see a Timeline in action.

A Timeline is broken down into a collection of keyframes. A *keyframe* is a point at which you want the action of your animation to change. For example, you are going to make an animation of a butterfly moving from the top of the screen to the bottom. Therefore, your keyframes will represent the start of your animation at the top of the screen as well as the end of your animation at bottom of the screen, as shown in Figures 8-1 and 8-2, respectively. The job of the Timeline is to fill in the space in between.

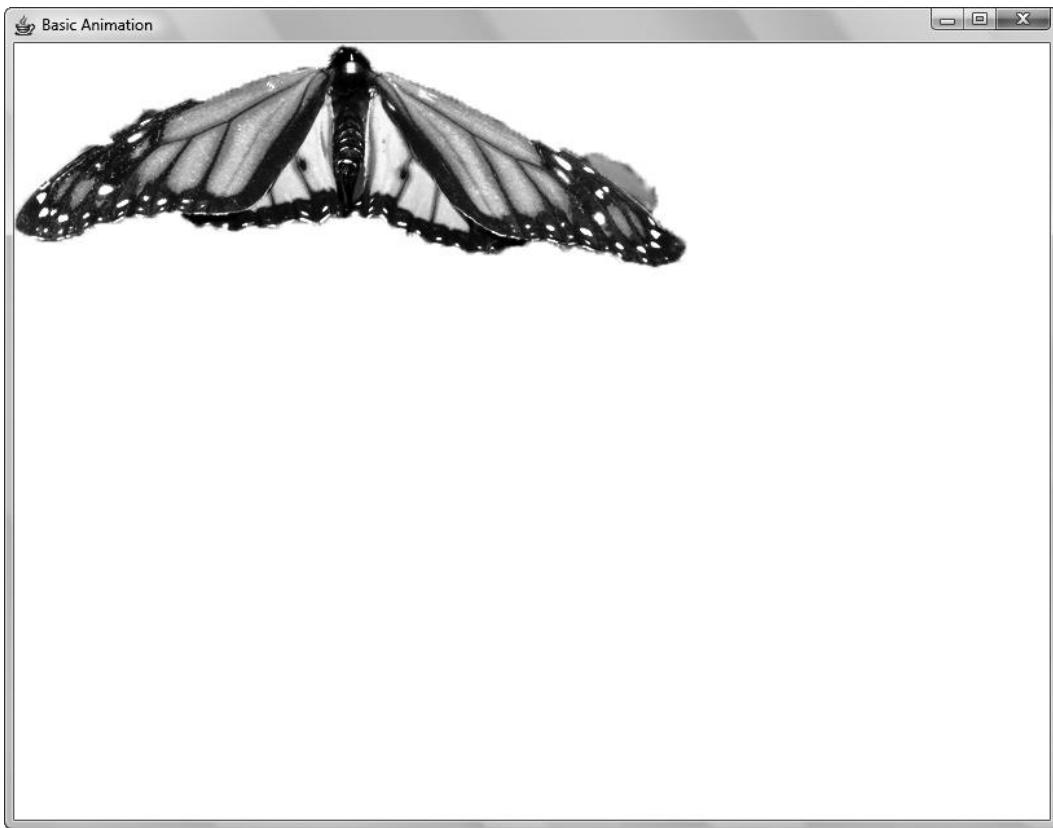
You are now going to animate the butterfly image used in the previous two chapters. You will make the butterfly image move down along the y axis. To begin, set up your images by calling both the background and the butterfly out of the `butterfly.fxz` file.

### NOTE

For a refresher on how to call images from an FXZ file, review Chapter 6.

The following code sets up your image variables:

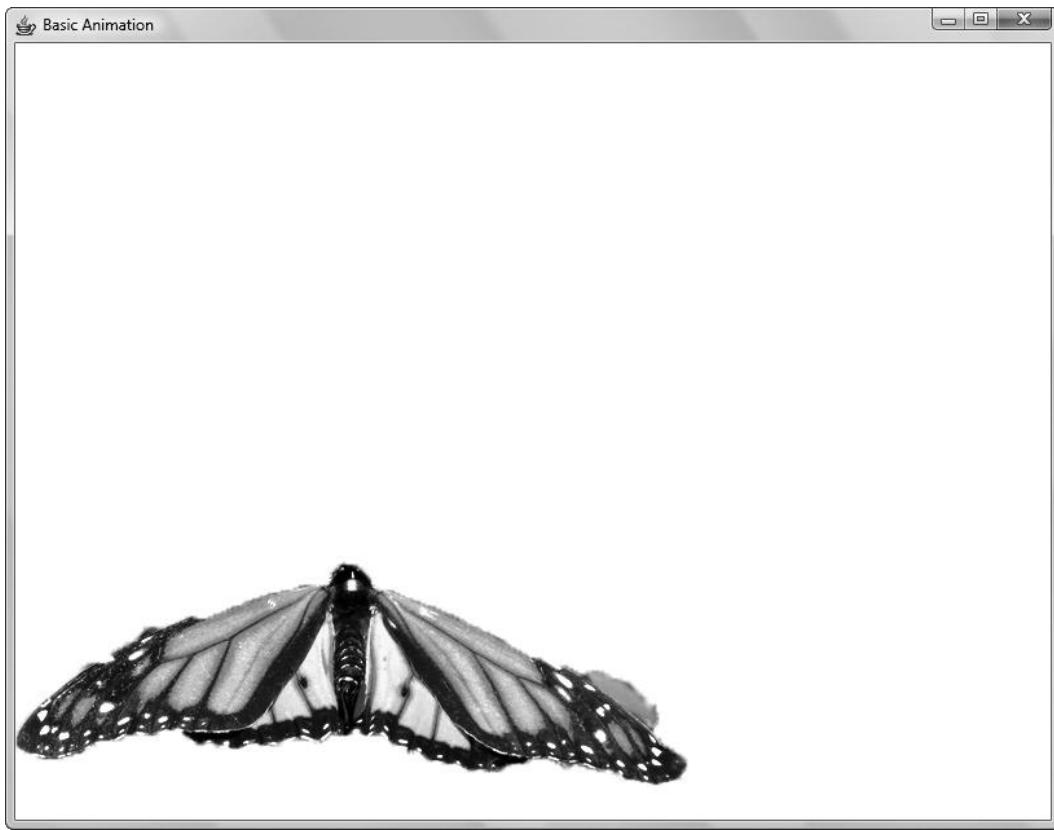
```
/*
 * Chapter8.fx
 *
 * v1.0 - J. F. DiMarzio
 */
```



**Figure 8-1** The first butterfly keyframe

```
* 5/27/2010 - created
*
* Basic Animation
*
*/
package com.jfdimarzio.javafxforbeginners;
import javafx.stage.Stage;
import javafx.fxml.FXDNode;
import javafx.scene.image.ImageView;
import javafx.scene.Group;

/**
 * @author JFDiMarzio
 */
```



**Figure 8-2** The second butterfly keyframe

```
/* Create image variables */
var imagePath : String = "{__DIR__}images/butterfly.fxz";
var butterflyImage : FXDNode = FXDNode{url: imagePath;}
var groupImage : Group = butterflyImage.getGroup("group1");
var butterfly : ImageView;
var background : ImageView;
/* END Create */

/* Set images */
butterfly = (butterflyImage.getNode("butterfly") as ImageView);
background = (butterflyImage.getNode("background") as ImageView);
/* End Set */

/* Move butterfly to y0 */
butterfly.y = 0;
var y: Integer = 0;
```

If you have read Chapters 6 and 7, you will recognize this code. Therefore, this is a quick overview. The first section of code contains the variable declarations. The five variables here hold your images, path, and group. Notice that the imagePath variable holds the {\_\_DIR\_\_} constant pointer to the FXZ file. This imagePath is used to build the butterflyImage, which in turn is used to build the groupImage.

The second section of code is used to build the butterfly and background images. These images are pulled from the butterflyImage using the getNode() function. Next, the y axis location of the butterfly image is set to 0. This places the butterfly at the top of the screen to give it room to move down. Finally, an integer named Y is created to be used later by the Timeline.

Once your images are set, create a Scene and assign groupImage to the content:

```
import javafx.scene.Scene;  
  
Stage{  
    title: "Basic Animation"  
    scene:  
        Scene{  
            width: 800  
            height: 600  
            content: [  
                groupImage  
            ]  
        }  
    }  
}
```

Now that the basic setup of the script is complete, you can begin to set up your Timeline. You will create a Timeline that moves the butterfly from the top of the screen (y:0) to the bottom (y:400) in two seconds. To start, create the Timeline shell:

```
Timeline{  
}  
}.play();
```

Notice the .play() call at the end of the Timeline. The function of this call is exactly as the name implies: It plays the Timeline to begin your action.

To use the Timeline, you will need to add a keyFrames collection to it. This collection will hold two keyframes. The first keyframe represents the butterfly at the top of the screen at zero seconds. The second keyframe represents the butterfly at the bottom of the screen after two seconds have elapsed. Here is a simplified example of keyFrames:

```
Timeline{  
    keyFrames: [  
        at (0s) {}  
    ]  
}
```

```
        at (2s) { }
    ]
}.play();
```

Now it is time to code the core of the Timeline. Within each keyframe you are going to move the butterfly by setting its translateY value. You are going to set translateY to 0 in the first keyframe, and to 400 in the second. The Timeline will do the rest for you. Here is the code:

```
import javafx.animation.Interpolator;

def frame = butterfly;
Timeline{
    keyFrames:[
        at (0s) { frame.translateY => 0 }
        at (2s) { frame.translateY => 400 tween Interpolator.LINEAR }
    ]
}.play();
```

The “tween” notation in the second keyframe tells the Timeline to build all the values between 0 and 400 and assign them to translateY over the two seconds.

### NOTE

In animation, the process of drawing all the images between two keyframes is known as  *tweening*.

The completed script should look as follows:

```
/*
 * Chapter8.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 5/27/2010 - created
 *
 * Basic Animation
 *
 */

package com.jfdimarzio.javafxforbeginners;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.fxml.FXDNode;
import javafx.scene.image.ImageView;
import javafx.animation.Timeline;
import javafx.animation.Interpolator;
import javafx.scene.Group;
```

```
/**  
 * @author JFDiMarzio  
 */  
  
/* Create image variables */  
var imagePath : String = "{__DIR__}images/butterfly.fxz";  
var butterflyImage : FXDNode = FXDNode{url: imagePath};  
var groupImage : Group = butterflyImage.getGroup("group1");  
var butterfly : ImageView;  
var background : ImageView;  
/* END Create */  
  
/* Set images */  
butterfly = (butterflyImage.getNode("butterfly") as ImageView);  
background = (butterflyImage.getNode("background") as ImageView);  
/* End Set */  
  
/* Move butterfly to y0 */  
butterfly.y = 0;  
  
def frame = butterfly;  
Timeline{  
    keyFrames:[  
        at (0s) { frame.translateY => 0}  
        at (2s) { frame.translateY => 400 tween Interpolator. LINEAR }  
    ]  
}.play();  
  
Stage{  
    title: "Basic Animation"  
    scene:  
        Scene{  
            width: 800  
            height: 600  
            content:[  
                groupImage  
            ]  
        }  
    }  
}
```

Run this script and you will see the butterfly image move from the top of the screen to the bottom.

What if you want the butterfly to keep moving up and down indefinitely? This is easy to do. First, to make the butterfly move back up after it has reached the bottom, you set the autoReverse property of the Timeline to true. Then, to make the process repeat, set the repeatCount parameter, as shown in the following code:

**TIP**

The repeatCount parameter can be set to a definite number such as 2 or 450. You can also set the repeatCount parameter to the constant INDEFINITE to repeat indefinitely.

```
Timeline{  
    repeatCount: Timeline.INDEFINITE;  
    autoReverse:true;  
    keyFrames:[  
        at (0s) { frame.translateY => 0 }  
        at (2s) { frame.translateY => 400 tween Interpolator.LINEAR }  
    ]  
}.play();
```

This process is good for simple motions, but what if you want to move your butterfly in a more elaborate way? The next section of this chapter covers animating your images along a path.

## Animating Along a Path

If you want to do a lot of math—and some tricky calculations—you can create a lot of movements with the animation style explained in the previous section. However, if you really want to do some complex animation, such as moving an object around the screen in a curving motion, you will want to use path animation, which is another method that JavaFX has for creating animation that allows you to move an object around a predefined path. In this section you learn how to create a path using knowledge you picked up in previous chapters. You then use this path to animate the butterfly.

The concept here is that you can create a path using lines, arcs, and points. JavaFX will then animate your image moving along this path.

To begin, set up your Chapter8.fx file as show here:

```
/*  
 * Chapter8.fx  
 *  
 * v1.0 - J. F. DiMarzio  
 *  
 * 5/27/2010 - created  
 *  
 * Basic Animation  
 */
```

```
package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;

import javafx.fxml.FXDNode;
import javafx.scene.image.ImageView;
import javafx.scene.Group;

/**
 * @author JFDiMarzio
 */

/* Create image variables */
var imagePath : String = "{__DIR__}images/butterfly.fxz";
var butterflyImage : FXDNode = FXDNode{url: imagePath;};
var groupImage : Group = butterflyImage.getGroup("group1");
var butterfly : ImageView;
var background : ImageView;
/* END Create */

/* Set images */
butterfly = (butterflyImage.getNode("butterfly") as ImageView);
background = (butterflyImage.getNode("background") as ImageView);
/* End Set */

/* Move butterfly to y0 */
butterfly.y = 0;

Stage {
    title : "MyApp"
    onClose: function () { }
    scene: Scene {
        width: 800
        height: 600
        content: [ groupImage ]
    }
}
```

Similar to what you have seen before, this code grabs the same butterfly and background images you have been working with and displays them to the Scene. In the previous section you animated the butterfly image travelling up and down the y axis. For this example you are going to animate the butterfly moving around an abstract path.

The next step is to create the path you want your butterfly to travel along. You will use a Path node to create this path. The node accepts a group of elements to create a path

from. You can easily create a group of elements that makes an abstractly shaped path. The following piece of code defines a small element array with some basic line-based shapes:

### NOTE

Be sure to take note of the required packages in the following code.

```
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.ArcTo;
import javafx.scene.shape.ClosePath;

/* Initial shape to move butterfly around */
def pathShape = [
    MoveTo {x : 150.0, y : 150.0}
    ArcTo {x : 350.0, y : 350.0, radiusX : 150.0, radiusY : 300.0}
    ArcTo {x : 150.0, y : 150.0, radiusX : 150.0, radiusY : 300.0}
    ClosePath{}
];

```

There is nothing too complex or tricky about what is happening here. You have created a collection of elements named pathShape. The elements contained within pathShape are MoveTo, two instances of ArcTo, and ClosePath. The combination of these elements creates a path your butterfly can follow.

The MoveTo element does exactly what the name suggests: It moves you to a specific point on the Cartesian grid. In this case it moves you to x150, y150. You are specifying this as your first element to cleanly move the starting point of your path before you start “drawing.”

The next two elements draw arcs. The first ArcTo element draws an arc from the last position of the point (in this case, x150, y150, thanks to the MoveTo element). The second ArcTo draws another arc from the end point of the last arc. Finally, the ClosePath element “connect the dots” and closes your path (if it does not close organically).

Keep in mind, all you have right now is a collection of elements. You do not yet have a path. Luckily, a path can be created from a group of elements. You will pass this group of elements into the definition of a path, as shown in the following code. Again, take care to note the package that needs to be imported for Path:

```
import javafx.scene.shape.Path;

/* Path definition */
def butterflyPath : Path = Path{
    elements: pathShape
}
```

In this code, you pass the pathShape into the elements of the path. This tells JavaFX that you want to turn your group of elements into a Path node. The Path node takes care of all the work needed to create a node from the group of elements you defined. The JavaFX animation package can now take this Path node and use it to animate your butterfly:

```
import javafx.animation.transition.PathTransition;
import javafx.animation.transition.OrientationType;
import javafx.animation.Timeline;
import javafx.animation.transition.AnimationPath;

/* Animation */
PathTransition{
    repeatCount: Timeline.INDEFINITE
    duration: 10s
    orientation: OrientationType.ORTHOGONAL_TO_TANGENT
    node: butterfly
    path: AnimationPath.createFromPath(butterflyPath)
}.play();
```

To create your animation, you use a PathTransition class, which takes in a few familiar parameters. Like Timeline, PathTransition can accept parameters for autoReverse, repeatCount, duration, and an interpolator. However, it is the node, path, and orientation that you want to focus on for this animation.

The node is the object you want animated. In this case, the butterfly image from the image group. The butterfly is assigned to the node in the PathTransition class.

### TIP

Because butterfly is a member of an image group, and the image group is being written to the content of the Scene, the animation of the butterfly will be displayed over the background.

The node will be animated along the path you created earlier. However, the path you created is a Path node, and PathTransition is expecting an AnimationPath. No need to worry, though: The createFromPath() function of AnimationPath will create an AnimationPath from a Path node. In the previous code sample, butterflyPath is passed into createFromPath() and the result is assigned to path.

Finally, the orientation parameter specifies the position of the node as it is animated along the path. If you do not specify an orientation, the image will remain in whatever orientation it is in when it is drawn to the screen. Therefore, in the case of the butterfly,

it will remain in the “heads-up” orientation as it travels around your elliptical path. However, you can also specify an orientation of ORTHOGONAL\_TO\_TANGENT. This is a constant that tells JavaFX to change the orientation of the node as it moves along the path. This change in orientation gives the animation a more organic feel.

The full path animation script should look as follows:

```
/*
 * Chapter8.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 5/27/2010 - created
 *
 * Basic Animation
 *
 */

package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.ArcTo;
import javafx.scene.shape.ClosePath;
import javafx.fxd.FXDNode;
import javafx.scene.image.ImageView;
import javafx.animation.Timeline;
import javafx.scene.Group;
import javafx.animation.transition.PathTransition;
import javafx.animation.transition.OrientationType;
import javafx.animation.transition.AnimationPath;
import javafx.scene.shape.Path;

/**
 * @author JFDiMarzio
 */

/* Create image variables */
var imagePath : String = "{__DIR__}images/butterfly.fxz";
var butterflyImage : FXDNode = FXDNode{url: imagePath;};
var groupImage : Group = butterflyImage.getGroup("group1");
var butterfly : ImageView;
var background : ImageView;
/* END Create */
```

```
/* Set images */
butterfly = (butterflyImage.getNode("butterfly") as ImageView);
background = (butterflyImage.getNode("background") as ImageView);
/* End Set */

/* Move butterfly to y0 */
butterfly.y = 0;

/* Initial shape to move butterfly around */
def pathShape = [
    MoveTo {x : 150.0, y : 150.0}
    ArcTo {x : 350.0, y : 350.0, radiusX : 150.0, radiusY : 300.0}
    ArcTo {x : 150.0, y : 150.0, radiusX : 150.0, radiusY : 300.0}
    ClosePath{}
];
/* Path definition */
def butterflyPath : Path = Path{
    elements: pathShape
}

/* Animation */
PathTransition{
    node: butterfly
    repeatCount: Timeline.INDEFINITE
    duration: 10s
    orientation: OrientationType.ORTHOGONAL_TO_TANGENT
    interpolate: true
    path: AnimationPath.createFromPath(butterflyPath)
}.play();

Stage {
    title : "MyApp"
    onClose: function () { }
    scene: Scene {
        width: 800
        height: 600
        content: [ groupImage ]
    }
}
```

Compile the preceding code and run it in the default profile. You will see the image of the butterfly animated around an elliptical path.

## Try This Create a Path Animation

In the previous chapters, the “Try This” sections have focused on added functionality that may not have been directly covered in the chapter. However, the skills covered in this chapter are so important that this section will focus on enhancing these skills.

Create a new project and add an image or shape to it. Then, try creating your own path along which you will animate the image. Experiment with paths of different sizes and lengths. Adjust the speed of your Timelines to change the feel of the animation.

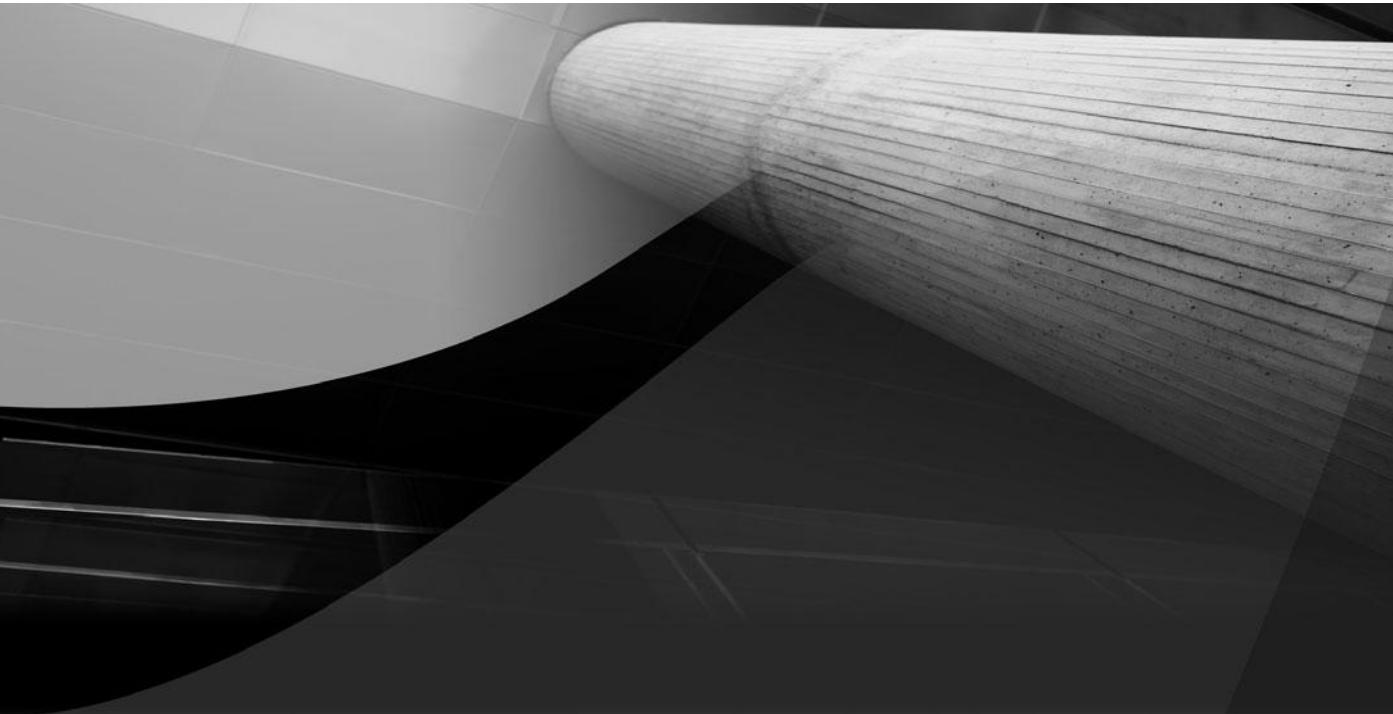
The more you are comfortable with the animation capabilities of JavaFX, the better your applications will be.



## Chapter 8 Self Test

1. Why is timing important to animation?
2. What controls the timer in JavaFX animation?
3. What does a Timeline take in as a parameter?
4. How do you start the Timeline?
5. True or false? The transition notation tells the Timeline to build all the values between the ones specified in your keyframes.
6. Which parameter sets the number of times a Timeline executes?
7. What is the purpose of ClosePath()?
8. A path is created from a group of what?
9. What function is used to create an AnimationPath from a Path node?
10. Which OrientationType will change the orientation of the node as it moves along the path?

*This page intentionally left blank*



# Chapter 9

## Using Events

## Key Skills & Concepts

- Reacting to mouse events
  - Using anonymous functions
  - Trapping key events
- 

This chapter introduces you to JavaFX events. The concepts that are required when working with events are key in producing rich, interactive applications. Events allow your application to respond to the actions of the user.

## What Are Events?

An Events provide a way to interact with your users in JavaFX. As a developer, you can use many types of events to enable your users to interact with and react to your environment. Events can be fired as a result of the user interacting in some way with your environment, and you can capture or trap these events and write actions around the information you have gathered.

In this chapter you discover and utilize two common sets of events: mouse events (`onMouse*`) and key events (`onKey*`).

### Mouse Events

A mouse event is the generic term for the action of any pointer-style input. The provider of this input does not necessarily have to be a physical mouse. Any device that moves a pointer across an environment will generate a mouse event. A mouse, stylus, track ball, track pad, and any other navigational tool will generate a mouse event if it moves your pointer around your JavaFX environment.

#### **NOTE**

Not all the mouse events will fire with input devices that are not a mouse. For example, `onMouseWheelMoved` will not be fired with a touch screen or stylus input.

**NOTE**

The following mouse events are specific to nodes, which means you must listen for these events as they pertain to a specific node in your Scene. Therefore, you cannot just randomly detect that the onMousePressed event was fired generally on your Scene; instead, you have to detect if onMousePressed is fired on a specific node such as a button or a test box.

Here's a list of the actions that you can trap using mouse events:

- onMouseClicked
- onMousePressed
- onMouseReleased
- onMouseDragged
- onMouseEntered
- onMouseExited
- onMouseMoved
- onMouseWheelMoved

While some of these actions may be named descriptively enough for you to figure out what action the listen to, some have similar and confusing names. Therefore, let's clear up any confusion by describing what each action is before we dive into building code around them.

**NOTE**

Keep in mind that although these actions may be part of the node in the common profile, how you deal with them may differ depending on the profile.

- **onMouseClicked** This event is fired if a full mouse click is detected (the user presses and releases the mouse button).
- **onMousePressed** This event fires when the user presses the mouse button. The user does not have to release the button to fire this event like onMouseClick.
- **onMouseReleased** This event is the second half of the onMouseClick event. This event fires when a mouse button is released.

- **onMouseDragged** This event fires when a mouse button is held down and the mouse is moved. Trapping this event will also help you to track the x and y axis values for the drag.
- **onMouseEntered** This event is fired when the mouse pointer enters the node to which the event is attached.
- **onMouseExited** This event is fired when the mouse pointer exits the node to which the event is attached.
- **onMouseMoved** This event is fired when the mouse is moved within the borders of the given node.
- **onMouseWheelMoved** This event is fired each time the mouse wheel is clicked.

Some of these events may look redundant. For example, shouldn't `onMousePressed + onMouseReleased = onMouseClicked`? Not necessarily. Let's look at a few scenarios.

If you click a node, you will see the following events fired in order:

- `onMousePressed`
- `onMouseClicked`

Notice that no `onMouseReleased` is fired. JavaFX recognizes the completion of the `onMousePressed` event as the logical conclusion of an `onMouseClicked` event. However, if you move the mouse between the time you press the mouse button and release it (that is, perform an intentional or unintentional mouse drag), you will fire the following events in order:

- `onMousePressed`
- `onMouseDragged`
- `onMouseReleased`

In this case, no `onMouseClicked` event is fired. Because other events are fired between the press and the release, JavaFX does not fire an `onMouseClicked` event.

The best way to understand this is that `onMouseClicked` is a specific event that encapsulates a contiguous pair of `onMousePressed` and `onMouseReleased` events. If any events are fired between `onMousePressed` and `onMouseReleased`, the action is no longer considered an `onMouseClicked` event.

Why then does `onMousePressed` still fire in both scenarios? Because JavaFX cannot predict what will happen after the `onMousePressed` event. It cannot say whether the action

will be an onMouseClicked or onMouseReleased event until the action is completed; therefore, the onMousePressed will always fire.

Now that you understand when each event should fire, let's discuss how you trap them.

The onMouse\* events are inherited through Node. Anything that inherits from Node will be able to trap these events. Therefore, all the shapes, buttons, labels, and text boxes that you can add to a Scene will trap these events and let you perform actions based on them.

### TIP

Create a new, empty JavaFX script named Chapter9.fx to try the following examples.

You should have a new, empty file that looks like this:

```
/*
 * Chapter9.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/15/2010 - created
 *
 * Using Events
 *
 */
```

```
package com.jfdimarzio.javafxforbeginners;

/***
 * @author JFDiMarzio
 */
```

Now, add a circle to the Scene. This circle will be the node we use to trap our mouse events. Finally, add a label to display the events. Your code should look like this:

### TIP

For more information about creating circles, refer to Chapter 4.

```
/*
 * Chapter9.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/15/2010 - created
 *
 * Using Events
 *
 */
```

```
package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.shape.Circle;
import javafx.scene.paint.Color;
import javafx.scene.control.Label;

/*
 * @author JFDiMarzio
 */

Stage {
    title : "UsingEvents"
    onClose: function () {   }
    scene: Scene {
        width: 450
        height: 400
        content: [
            Circle {
                centerX: 100,
                centerY: 100
                radius: 100
                fill: Color.BLACK
            }
            Label {
                text: "";
                layoutX:10;
                layoutY:250;
            }
        ]
    }
}
```

Set up a variable that your label can bind to. You will write out the contents of your events to this variable, which will be bound directly to the label's text property:

```
var message : String = "";
```

You can now write an anonymous function to handle each mouse event. The purpose of the anonymous function is to immediately perform an action when the event is fired. The function will encapsulate all the logic you want to perform when a specific event is fired. In this example, your anonymous function will write the contents of the event to the message variable, which is bound to the label text.

The anonymous function will need to take in a MouseEvent as a parameter. Import the following package to use MouseEvent:

```
import javafx.scene.input.MouseEvent;
```

### **NOTE**

Two instances of MouseEvent appear in different packages. There is another MouseEvent in the java.awt package. You will receive errors in this example if you import the wrong one.

Now you can set up a simple function to write the event to a variable, as seen in the following code sample:

```
function (event : MouseEvent) : Void{  
    message = event.toString();  
}
```

Use this function in each of the mouse events you want to trap. The contents of the event will then be written out to the label on the screen.

You can start with the onMousePressed event of the circle node that you added to the Scene:

```
onMousePressed: function (event : MouseEvent) : Void{  
    message = event.toString();  
}
```

The logic here is that when onMousePressed is triggered—that is, when the user presses the mouse button within the bounds of the circle node—the event is captured and written out to the message variable. This variable, which is bound to the label text, is written to the screen just below the circle.

Replicate this same anonymous function for the onMouseReleased and onMouseClicked events of the circle node. Your finished code should look like this:

```
/*  
 * Chapter9.fx  
 *  
 * v1.0 - J. F. DiMarzio  
 *  
 * 6/15/2010 - created  
 *  
 * Using Events  
 *  
 */
```

```
package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.shape.Circle;
import javafx.scene.paint.Color;
import javafx.scene.control.Label;
import javafx.scene.input.MouseEvent;

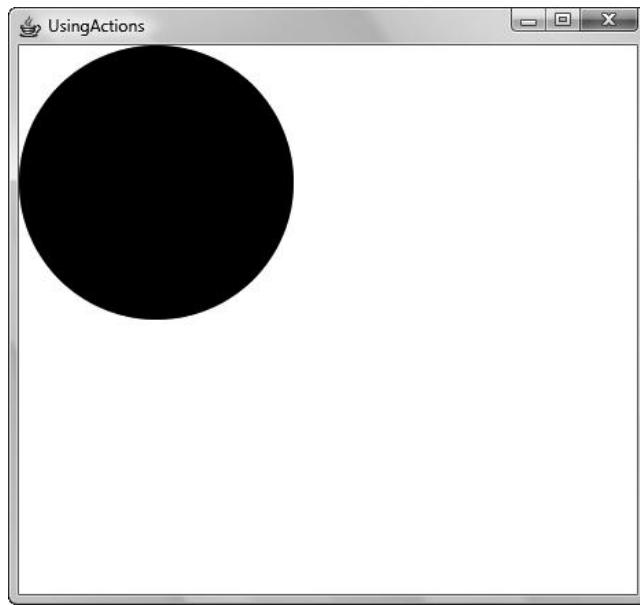
/**
 * @author JFDiMarzio
 */
var message : String = "";

Stage {
    title : "UsingEvents"
    onClose: function () { }
    scene: Scene {
        width: 450
        height: 400
        content: [
            Circle {
                centerX: 100,
                centerY: 100
                radius: 100
                fill: Color.BLACK
                onMousePressed: function (event : MouseEvent) : Void{
                    message = event.toString();
                }
                onMouseReleased:function (event : MouseEvent) : Void{
                    message = event.toString();
                }
                onMouseClicked:function (event : MouseEvent) : Void{
                    message = event.toString();
                }
            }
            Label {
                text: bind message;
                layoutX:10;
                layoutY:250;
            }
        ]
    }
}
```

Run the code in the default profile. You will see a black circle on the screen like the one shown in Figure 9-1.

Now, move your mouse into the circle and click the mouse button. This will fire the onMousePressed event of the circle and write the following information to the screen:

```
MouseEvent [x=129.0, y=92.0, button=PRIMARY PRESSED]
```



**Figure 9-1** A black circle

Release the mouse button without moving it. You should see the following information:

```
MouseEvent [x=129.0, y=92.0, button=PRIMARY CLICKED]
```

This indicates that the onMouseClicked event has fired.

Let's perform another test. Try pressing down the mouse button inside the circle, dragging the mouse a few pixels, and then releasing it. You should see the following information display:

### **NOTE**

Both lines will not display at the same time; instead, they display sequentially.

```
MouseEvent [x=86.0, y=89.0, button=PRIMARY PRESSED]
MouseEvent [x=133.0, y=129.0,
           button=PRIMARY, dragX 63.0, dragY 27.0 RELEASED]
```

Notice that the x and y coordinates of the drag are recorded in the press and release. Using this information, you can tell how far an item was dragged, what direction it was dragged in, and where it was released.

Try adding the remaining mouse events to the script (shown next) and testing different scenarios. You should see a large amount of very useful information that you can trap and act upon in your scripts.

```
/*
 * Chapter9.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/15/2010 - created
 *
 * Using Events
 *
 */

package com.jfdimizaro.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.shape.Circle;
import javafx.scene.paint.Color;
import javafx.scene.control.Label;
import javafx.scene.input.MouseEvent;

/**
 * @author JFDiMarzio
 */
var message : String = "";

Stage {
    title : "UsingEvents"
    onClose: function () { }
    scene: Scene {
        width: 450
        height: 400
        content: [
            Circle {
                centerX: 100,
                centerY: 100
                radius: 100
                fill: Color.BLACK
                onMousePressed: function (event : MouseEvent) : Void{
                    message = event.toString();
                }
                onMouseReleased: function (event : MouseEvent) : Void{
                    message = event.toString();
                }
                onMouseClicked: function (event : MouseEvent) : Void{
                    message = event.toString();
                }
            }
        ]
    }
}
```

```
onMouseDragged:function (event : MouseEvent) : Void{
    message = event.toString();
}
onMouseEntered:function (event : MouseEvent) : Void{
    message = event.toString();
}
onMouseExited:function (event : MouseEvent) : Void{
    message = event.toString();
}
onMouseWheelMoved:function (event : MouseEvent) :
Void{
    message = event.toString();
}

}
Label {
    text: bind message;
    layoutX:10;
    layoutY:250;
}
]
}
```

**TIP**

Keep in mind that the `onMouse*` events are inherited from `Node`. The major implication of this is that `Stage` and `Scene` do not inherit from `Node`. Therefore, you can only trap these events as they occur within `Node` on the `Scene`. You cannot trap these events if they happen randomly within the context of your `Scene`.

Another type of event you can trap is a key event. The following section goes over how to trap and use key events.

## Key Events

A key event, as the name suggests, is fired when the user interacts with the keyboard. Capturing these keyboard events can be very useful. For example, you could write a script that allows the user to move objects around the `Scene` with the arrow keys or to stop some animation using the `esc` key. Either way, trapping key events can be very useful in your JavaFX scripts.

The key events are also inherited from `Node`. Therefore, like the `onMouse*` events, only nodes can trap key events. The three `onKey*` events are `onKeyPressed`, `onKeyReleased`,

and onKeyTyped. When a user interacts with the keyboard (or keypad for mobile users), the events are fired in this order:

- onKeyPressed
- onKeyTyped
- onKeyReleased

Modify your script from the last section to trap these events. The code in your script should look like this when you are finished:

```
/*
 * Chapter9.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/15/2010 - created
 *
 * Using Events
 *
 */

package com.jfdimizaro.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.shape.Circle;
import javafx.scene.paint.Color;
import javafx.scene.control.Label;
import javafx.scene.input.KeyEvent;

/**
 * @author JFDiMarzio
 */
var message : String = "";

Stage {
    title : "UsingEvents"
    onClose: function () { }
    scene: Scene {
        width: 450
        height: 400
        content: [
            Circle {
                centerX: 100,
                centerY: 100
            }
        ]
    }
}
```

```
        radius: 100
        fill: Color.BLACK
        focusTraversable: true;
        onKeyPressed:function(event: KeyEvent):Void{
            message = event.toString();
        }
        onKeyReleased:function(event: KeyEvent):Void{
            message = event.toString();
        }
        onKeyTyped:function(event: KeyEvent):Void{
            message = event.toString();
        }
    }
    Label {
        text: bind message;
        layoutX:10;
        layoutY:250;
    }
}
}
```

If you look carefully at the code, you will notice one slight difference in the properties of the Circle node. To get this example to function correctly, a property named focusTraversable needs to be set. The focusTraversable property lets the circle accept the focus of the Scene's action. Without this property being set, JavaFX would not be able to tell what node you are interacting with when you type on the keyboard.

### TIP

The focusTraversable property also allows focus to be passed between nodes. If you need a user to be able to move focus from node to node, set focusTraversable to true on both nodes.

Run this script using the Mobile profile. It should run as shown in Figure 9-2.

With the script running in the mobile emulator, press the number 2 on the keypad. You should see the following:

```
KeyEvent node [javafx.scene.shape.Circle] ,
              char [] , text [2] , code [VK_2]
KeyEvent node [javafx.scene.shape.Circle] ,
              char [2] , text [2] , code [VK_UNDEFINED]
KeyEvent node [javafx.scene.shape.Circle] ,
              char [] , text [2] , code [VK_2]
```



**Figure 9-2** Trapping keys in the Mobile profile

Take notice of the char[] property of the KeyEvent. This property is only available on onKeyTyped. Therefore, if you are trapping just the char[] property, you will only see it on onKeyTyped. Now press the up arrow in the emulator and compare the outputs:

```
KeyEvent node [javafx.scene.shape.Circle] , char [] , text [Up] , code [VK_UP]
KeyEvent node [javafx.scene.shape.Circle] , char [] , text [Up] , code [VK_UP]
```

Notice that the directional arrows in the mobile profile do not fire the onKeyTyped event. They will only fire the onKeyPressed and onKeyReleased events.

### TIP

In the event you cannot see the full event message printed on the screen of the mobile emulator, add the following line to your script. With this line in place, the event message will print out to the Output window in NetBeans:

```
System.out.println(message)
```

It is very important to note what events are fired and in what order. This will make your job easier when you are trying to trap and react to user input.

## Ask the Expert

**Q: What is the importance of the onMouse and onKey events?**

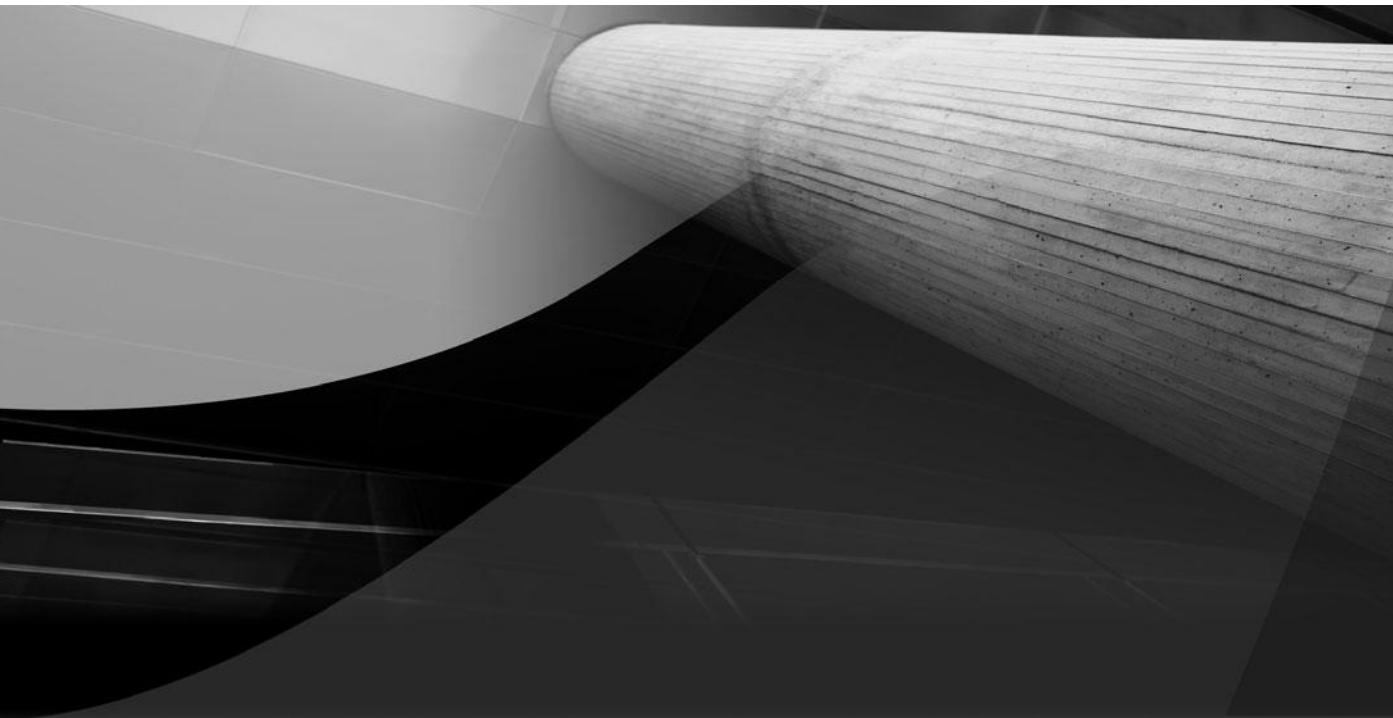
**A:** Using the events described in this chapter is one of the best ways you have to let the user interact with your application. Since the birth of computer applications, users have had two methods for interacting with them: a keyboard and a pointer. Inevitably, you will need to create an application that requires some level of user interaction. Being able to react to onMouse and onKey events is one of your greatest tools for gathering information from your users.



## Chapter 9 Self Test

1. Where are the onMouse\* events inherited from?
2. When is onMouseEntered fired?
3. True or false? The onMouseReleased event only fires when the mouse is dragged.

4. True or false? Anything that inherits from Node can trap onMouse\* events.
5. When events are used, what is the purpose of an anonymous function?
6. Which mouse event is fired when the mouse pointer exits the node to which the event is attached?
7. What three events are fired when the user interacts with the keyboard?
8. In what order are the key events fired?
9. What property will allow a node to accept focus?
10. True or false? The navigational buttons on a mobile phone will fire the onKeyTyped event.



# Chapter 10

## Give It Some Swing

## Key Skills & Concepts

- Understanding swing
  - Adding swing to a JavaFX script
  - Using SwingComboBoxItem
- 

**S**wing is a common package used in Java development. However, swing is also available to users of JavaFX. In this chapter you will learn how to use swing and swing components in your JavaFX applications.

## What Is Swing?

The `javafx.ext.swing` package contains a host of GUI (graphical user interface) tools. These tools can be used to build a useful interface for your JavaFX application. If you are looking to build a more business-like application using JavaFX, you will want to look at the available swing components.

If you are already an experienced Java developer, swing should be familiar to you. Swing has been available to Java developers since the early days of Java development. Java developers can use their knowledge of swing to jump right into JavaFX swing development. However, if you are an experienced Java developer who has worked with swing before, you will notice some differences between the full swing package that is provided to Java and that which is provided to JavaFX.

The version of the swing package provided with JavaFX does not include all the controls in the full Java package. The JavaFX version of swing only contains a subset of controls, including the following:

- SwingButton
- SwingCheckBox
- SwingComboBox
- SwingComboBoxItem

- SwingIcon
- SwingLabel
- SwingList
- SwingListItem
- SwingRadioButton
- SwingScrollPane
- SwingSlider
- SwingTextField
- SwingToggleButton
- SwingToggleGroup

All the components listed here for the swing package either directly or indirectly inherit from Node. This means you can use a swing component wherever you would use any other JavaFX node. In the following sections you learn about where and how you can use JavaFX swing components to add style and interactivity to your applications.

#### **NOTE**

Because the JavaFX swing components inherit from Node, all the `onMouse*` and `onKey*` events you learned about in Chapter 9 are available.

Let's take a look at the different swing components available to you within JavaFX.

## Swing Components

This section describes how to use the swing components available to you through the `javafx.ext.swing` package. These components add usability and interactivity to your applications. Everyone is familiar with the look and function of buttons, check boxes, and lists in an application. Using swing will let you add these familiar items to your JavaFX applications.

If you are creating business applications, such as surveys, look-up tools, or input forms, the swing components discussed in this section will help you.

To begin, create a new, empty JavaFX script and name it **Chapter10.fx**. The contents of this script should look as follows:

**TIP**

If you have just followed the examples in the previous chapter, your version of NetBeans may still be set up to execute code in the mobile emulator. Change your Run profile back to the default before running the scripts in this chapter.

```
/*
 * Chapter10.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/20/2010 - created
 *
 * Give it some Swing
 *
 */

package com.jfdimarzio.javafxforbeginners;

/**
 * @author JFDiMarzio
 */

// place your code here
```

In the following subsections you add four different swing components to this script. These four different components—`SwingButton`, `SwingCheckBox`, `SwingComboBox`, and `SwingComboBoxItem`—best represent the overall structure of the swing components. Learning and mastering these four components will help you understand all of them.

Let's start with the `SwingButton` component.

## SwingButton

`SwingButton` is a graphic button you can add to your script. This button allows users to interact with your applications by making selections and choosing actions. Everyone who uses a computer, cell phone, or most any other electronic device is familiar with the use of an onscreen button. Therefore, adding buttons to your site will give it a sense of familiarity and ease of use.

For this example you will be creating a button that increments a value every time it is clicked. That value will then be written to a `Text` node.

Set up an Integer variable named `clickValue` and a Text node that binds to it, as follows:

```
/*
 * Chapter10.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/20/2010 - created
 *
 * Give it some Swing
 *
 */

package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.text.Text;
import javafx.scene.text.Font;

/**
 * @author JFDiMarzio
 */

var clickValue : Integer = 0;

Stage {
    title : "UsingSwing"
    onClose: function () { }
    scene: Scene {
        width: 200
        height: 200
        content: [Text {
            font : Font {
                size: 24
            }
            x: 10, y: 50
            content: bind "Value = {clickValue}"
        }]
    }
}
```

If you compile and run this script now, it should look like Figure 10-1.



**Figure 10-1** App before swing is added

### TIP

String interpolation is great way to build functionality directly into your string values. The JavaFX string interpolation operators are the curly braces ({}). You can put variable and other functional script within these operators to change the contents of your string. In this example, you are interpolating the clickValue variable within your string. The bind keyword ensures that when clickValue changes, the string is updated accordingly.

Before you add a SwingButton to your script, you must import the swing package:

```
javafx.ext.swing.SwingButton
```

After adding the SwingButton to your Scene's content, you need to set up two properties: text and action. The text property defines the text that is displayed in your button. The name you provide for your button should be descriptive enough so that an uninformed observer can determine the function of the button. This can be something simple, such as “Submit” or “Calculate,” or it can describe something more complex, such as “Retrieve related records” or “Process schedule changes.”

```
SwingButton {  
    text: "Increment Value"  
}
```

The action property contains all the information the button needs to perform a function when the user clicks it. Like the onMouse\* and onKey\* events that are inherited from Node, the action property can take an anonymous function to encapsulate the code needed to execute an action. You can also create a function elsewhere in your script and call it from the action property. Either method is acceptable and will ultimately yield the same result.

For this example, use an anonymous function to increment the value of clickValue every time the button is clicked:

```
SwingButton {  
    text: "Increment Value"  
    action:function ():Void {  
        clickValue ++ ;  
    }  
}
```

**TIP**

The double plus sign operator, `++`, will increment a variable by 1. It is equivalent to `clickValue = clickValue + 1`.

Your finished script should look like this:

```
/*  
 * Chapter10.fx  
 *  
 * v1.0 - J. F. DiMarzio  
 *  
 * 6/20/2010 - created  
 *  
 * Give it some Swing  
 */  
  
package com.jfdimarzio.javafxforbeginners;  
  
import javafx.stage.Stage;  
import javafx.scene.Scene;  
import javafx.ext.swing.SwingButton;  
import javafx.scene.text.Text;  
import javafx.scene.text.Font;  
  
/**  
 * @author JFDiMarzio  
 */  
  
var clickValue : Integer = 0;  
  
Stage {  
    title : "UsingSwing"  
    onClose: function () { }  
    scene: Scene {  
        width: 200  
        height: 200
```

```
content: [SwingButton {
    text: "Increment Value"
    action:function ():Void {
        clickValue ++
    }
}
Text {
    font : Font {
        size: 24
    }
    x: 10, y: 50
    content: bind "Value = {clickValue}"
}
]
```

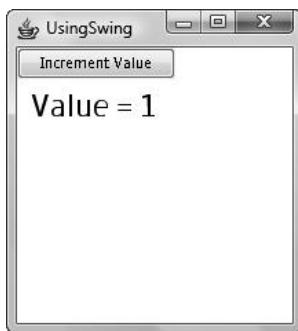
}

Compile and run this script. After clicking the button you will see the value increment, as shown in Figure 10-2.

### **NOTE**

This example may produce warnings from the compiler. These warnings will not prevent you from running the example and can be ignored here.

This example was a little basic. It explains in a concise way how to establish a basic rectangular button that performs an action. However, what if you want to do something a little out of the ordinary? The clip property is very useful for adding something extra to the look of your SwingButton.



---

**Figure 10-2** Using a SwingButton to increment a value

**NOTE**

The clip property is also inherited from Node, so anything that inherits from Node can utilize clip.

The purpose of the clip property is to allow you to define a clip area for your button to change the shape of the button. The clip property takes in a node and uses that node to reshape the button. Therefore, you can assign a circle to the clip property of a SwingButton and the result will be a round button.

Take a look at the following script. The text of the SwingButton has been changed, and the clip property has been defined as a circle:

```
/*
 * Chapter10.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/20/2010 - created
 *
 * Give it some Swing
 *
 */

package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.ext.swing.SwingButton;
import javafx.ext.swing.SwingSlider;
import javafx.scene.text.Text;
import javafx.scene.text.Font;
import javafx.scene.shape.Circle;

/**
 * @author JFDiMarzio
 */
var clickValue : Integer = 0;

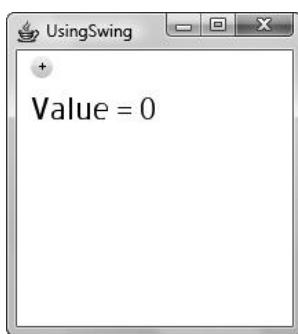
Stage {
    title : "UsingSwing"
    onClose: function () {   }
    scene: Scene {
        width: 200
        height: 200
    }
}
```

```
content: [SwingButton {
    text: "+"
    action:function ():Void {
        clickValue ++
    }
    clip: Circle {
        centerX: 18, centerY: 12
        radius: 8
    }
}
Text {
    font : Font {
        size: 24
    }
    x: 10, y: 50
    content: bind "Value = {clickValue}"
}
]
}
```

Compile and run this script. You will now see a round button with a plus sign in it that increments the value from the last example. The result is shown in Figure 10-3.

The clip property provides a very useful way to modify the look of your control. Take note that the bounds of the button have also been modified, not just the look of the button. The rectangular portion of the button that lies outside the clip region is no longer active or actionable.

Next, you learn about the SwingCheckBox component.



---

**Figure 10-3** A round button using clip

## CheckBox

In this section you set up a CheckBox. However, the process you use to set it up is slightly different from what you have used so far in this book. In an effort to introduce you to different ways of setting up and utilizing nodes, we are going to take a different approach in setting up the CheckBox.

You should start out with a basic script containing a Stage, Scene, and Text node. The Text node is used to display the selected value of the CheckBox. Your script should appear as follows:

```
/*
 * Chapter10.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/20/2010 - created
 *
 * Give it some Swing
 *
 */

package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.text.Text;
import javafx.scene.text.Font;

/**
 * @author JFDiMarzio
 */

Stage {
    title : "UsingSwing"
    onClose: function () {    }
    scene: Scene {
        width: 200
        height: 200
        content: [
            Text {
                font : Font {
                    size: 24
                }
                x: 10, y: 50
                content: bind "Value = {}"
            }
        ]
    }
}
```

The next step is to create a variable that is typed as a SwingCheckBox:

```
import javafx.ext.swing.SwingCheckBox;

def sampleCheckBox : SwingCheckBox = SwingCheckBox{  
    };
```

Set the text property of the SwingCheckBox to “Check Me”:

```
def sampleCheckBox : SwingCheckBox = SwingCheckBox{  
    text:"Check Me"  
    };
```

Finally, bind the string interpolator in the Text content to the selected property of the SwingCheckBox:

```
Text {  
    font : Font {  
        size: 24  
        }  
    x: 10, y: 50  
    content: bind "Value = {sampleCheckBox.selected}"  
}
```

The script you have written will create a check box. The selected property of the check box is a Boolean that indicates whether the check box has been checked. Because you have bound the value of the selected property to the Text content, you will see the text True or False when you check and uncheck the SwingCheckBox. The finished script shows where to add the sampleCheckBox to have it display to the screen.

The finished script should appear as follows:

```
/*  
 * Chapter10.fx  
 *  
 * v1.0 - J. F. DiMarzio  
 *  
 * 6/20/2010 - created  
 *  
 * Give it some Swing  
 */  
  
package com.jfdimarzio.javafxforbeginners;  
  
import javafx.stage.Stage;  
import javafx.scene.Scene;  
import javafx.scene.text.Text;
```

```
import javafx.scene.text.Font;
import javafx.ext.swing.SwingCheckBox;

/**
 * @author JFDiMarzio
 */
def sampleCheckBox : SwingCheckBox = SwingCheckBox{
    text:"Check Me"
};

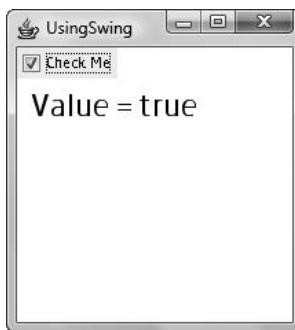
Stage {
    title : "UsingSwing"
    onClose: function () { }
    scene: Scene {
        width: 200
        height: 200
        content: [sampleCheckBox,
            Text {
                font : Font {
                    size: 24
                }
                x: 10, y: 50
                content: bind "Value = {sampleCheckBox.selected}"
            }
        ]
    }
}
```

Compile and run your script. Figures 10-4 and 10-5 illustrate the unchecked and checked states, respectively, of your script.

Next, you learn about SwingComboBox and SwingComboBoxItem.



**Figure 10-4** Unchecked SwingCheckBox with text



**Figure 10-5** Checked SwingCheckBox with text

## SwingComboBox and SwingComboBoxItem

SwingComboBox allows you to offer choices to your users and react on those choices. Most people are very familiar with the look, feel, and function of a combo box–style control. When a user is presented with a choice of different items, that choice is normally done through a combo box.

When using swing, you use two controls to create a functional combo box–style control. You need to use a SwingComboBox and one or many SwingComboBoxItem(s). Using these two swing controls together will give you a functional combo box.

Let's create a functional SwingComboBox that populates a Text node with the text of whatever item you select.

Create an empty script that contains just a Text node, as follows:

```
/*
 * Chapter10.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/20/2010 - created
 *
 * Give it some Swing
 *
 */

package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.text.Text;
import javafx.scene.text.Font;
```

```

/**
 * @author JFDiMarzio
 */

Stage {
    title : "UsingSwing"
    onClose: function () { }
    scene: Scene {
        width: 200
        height: 200
        content: [
            Text {
                font : Font {
                    size: 24
                }
                x: 10, y: 50
                content: bind "Value = { }"
            }
        ]
    }
}

```

Again, you should notice that the content of the Text node is set to an interpolator. This is where you bind the text of the selected item from the combo box.

In this script you are going to create a combo box that has three choices: One is blank (the default), one reads “Item A,” and the last reads “Item B.” The first step is to create a var of a SwingComboBox, as follows:

```

import javafx.ext.swing.SwingComboBox;

var sampleSwingComboBox : SwingComboBox = SwingComboBox {
}

```

The var, sampleSwingComboBox, right now is just a shell of a combo box; it will not do anything without some SwingComboBoxItems. The items property within the SwingComboBox will hold all your SwingComboBoxItems.

Create a SwingComboBoxItem as follows:

### **NOTE**

Keep in mind that the import statements are shown with the code for your convenience. However, in your script, all the import statements should appear together. The full script files illustrate this correctly.

```

import javafx.ext.swing.SwingComboBoxItem;

SwingComboBoxItem {
    text: "Item A"
}

```

Pretty easy, right? Now create three SwingComboBoxItems and assign them to the items property of the SwingComboBox:

```
var sampleSwingComboBox : SwingComboBox = SwingComboBox {
    items: [
        SwingComboBoxItem {
            text: ""
            selected: true
        }
        SwingComboBoxItem {
            text: "Item A"
        }
        SwingComboBoxItem {
            text: "Item B"
        }
    ]
}
```

### **NOTE**

One of the items in this code sample contains a selected property that is set to true. This property tells JavaFX that this item is the default item to be shown as selected in the combo box.

Finally, you need to find a way to set the contents of the Text node to the text of the selected combo box item. Luckily for you, there is a property of the SwingComboBox called selectedItem. This property contains the selected SwingComboBoxItem. Therefore, set the interpolator to selectedItem.text. The finished script should appear as follows and shows you where to add the ComboBox to the content of the Scene:

```
/*
 * Chapter10.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/20/2010 - created
 *
 * Give it some Swing
 *
 */

package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.text.Text;
```

```
import javafx.scene.text.Font;
import javafx.ext.swing.SwingComboBox;
import javafx.ext.swing.SwingComboBoxItem;

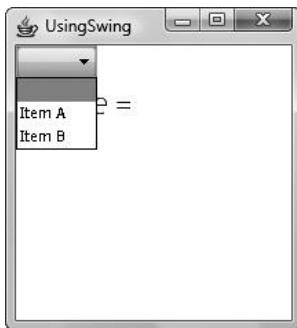
/**
 * @author JFDiMarzio
 */

var sampleSwingComboBox : SwingComboBox = SwingComboBox {
    items: [
        SwingComboBoxItem {
            text: ""
            selected: true
        }
        SwingComboBoxItem {
            text: "Item A"
        }
        SwingComboBoxItem {
            text: "Item B"
        }
    ]
}

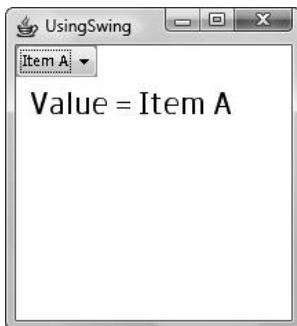
Stage {
    title : "UsingSwing"
    onClose: function () { }
    scene: Scene {
        width: 200
        height: 200
        content: [sampleSwingComboBox,
            Text {
                font : Font {
                    size: 24
                }
                x: 10, y: 50
                content: bind "Value =
{sampleSwingComboBox.selectedItem.text}"
            }
        ]
    }
}
```

Figures 10-6 and 10-7 show the SwingComboBox opened and closed, respectively.

Take some time to further explore the different properties of these swing components. Swing components can add a level of flexibility and usefulness to your JavaFX applications.



**Figure 10-6** SwingComboBox opened



**Figure 10-7** SwingComboBox closed

### Try This

## Create an Application with Swing

Swing is a very common Java element type. Even if you move on from JavaFX to other Java-based development platforms, chances are you will run into swing. The more you know about swing and the more comfortable you are with using it, the better your applications will be.

Create a new project and build a new Scene. Create your own application using only swing nodes. Add a TextBox and a Button to the Scene. Place the nodes around the Scene in the most logical position. For an added challenge, write some code behind the Button so that when it is clicked, the TextBox is populated with the words “You clicked my button.”

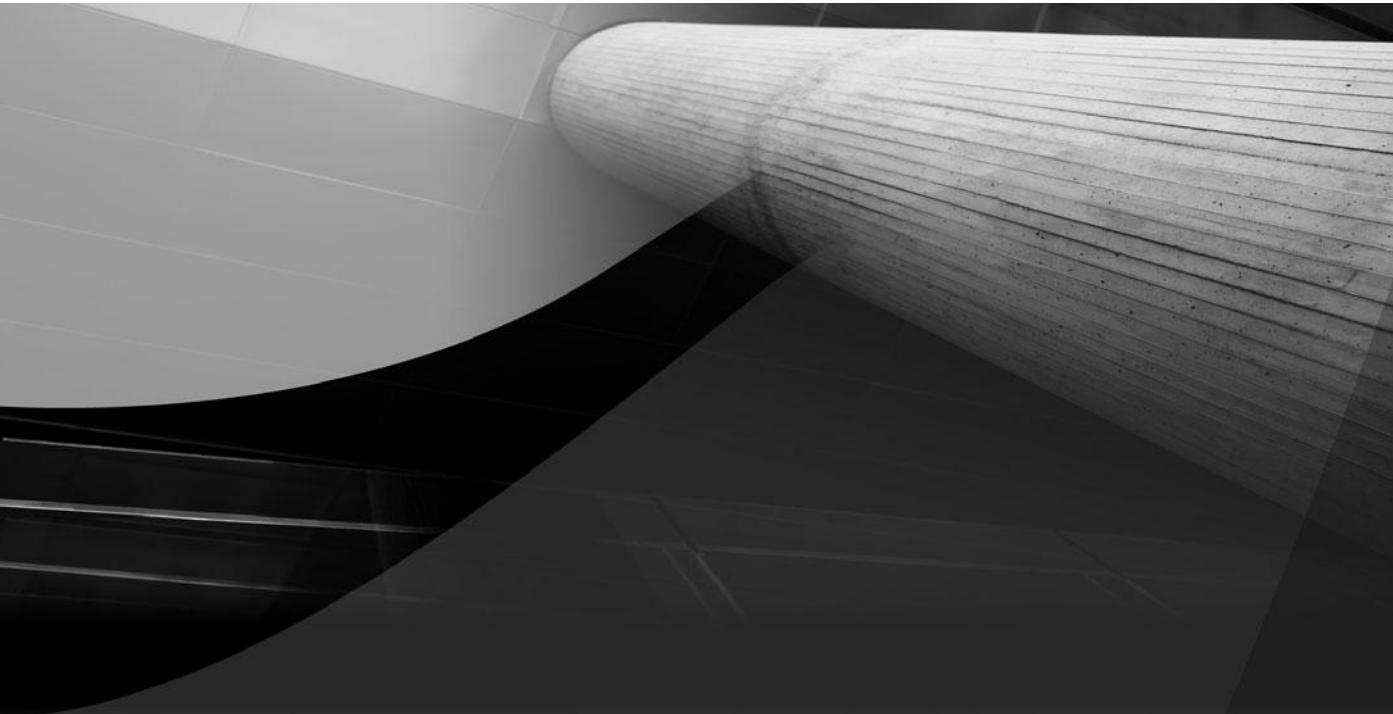
---



## Chapter 10 Self Test

1. What package contains the swing components for JavaFX?
2. True or false? The swing JavaFX package contains all the components available in Java.
3. Are the onMouse\* and onKey\* events available to swing components?
4. What are the JavaFX string interpolator operators?
5. What property of the SwingButton can hold an anonymous function that will execute when the button is clicked?
6. What property of the SwingButton can be used to change the shape of the button?
7. True or false? The isChecked property of the SwingCheckBox will tell you if the box is checked.
8. What swing component is used to populate SwingComboBox?
9. How do you set a SwingComboBoxItem to the default choice?
10. What property of SwingComboBox will tell you what SwingComboBoxItem has been selected?

*This page intentionally left blank*



# Chapter 11

## Custom Nodes and Overriding

## Key Skills & Concepts

- Implementing Node
  - Creating a custom node
  - Overriding default Node properties
- 

JavaFX comes with many great tools. Throughout this book you have learned about shapes, effects, and swing components—to name a few. However, what if you find that the nodes in JavaFX do not quite fit what you need? There is another way to get exactly what you need from JavaFX: You can always make a node that does exactly what you need. This can be by overriding an existing node or by creating an entirely new one. In this chapter you learn about both methods for getting precisely what you need out of JavaFX.

First, let's discuss overriding an existing node.

## Overriding a Node

Before you can learn about overriding a node, a critical question needs to be answered. What is overriding?

Classes, such as Node and SwingComboBox, contain methods and properties. You as a developer can take these methods and properties and change their default actions to whatever you want. For example, let's say you have a Dog class with a method called displayBreed(), as shown here:

```
public class Dog{  
    public function displayBreed(){  
        println("Chihuahua");  
    }  
}
```

You can extend this class and override the displayBreed() method to say something different, as follows:

```
public class MyDog extends Dog{  
    override function displayBreed(){  
        println("Elkhound");  
    }  
}  
public class YourDog extends Dog{  
}
```

In this example, a call to Dog.getBreed() would print the line “Chihuahua.” A call to YourDog.getBreed() would also print “Chihuahua.” However, because MyDog overrides the getBreed() method, a call to MyDog.getBreed() would print “Elkhound.”

Using overriding, you can create nodes with new or expanded functionality. In the following section you create a new SwingButton using overrides. To prepare for this exercise, create two new, empty JavaFX scripts. Name the first file **Chapter11.fx**. It should appear as follows:

```
/*
 * Chapter11.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/23/2010 - created
 *
 * Custom Nodes and Overriding
 *
 */
```

```
package com.jfdimarzio.javafxforbeginners;

/***
 * @author JFDiMarzio
 */
```

Name the second file **RoundButton.fx**. It should appear as follows:

**TIP**

It is important to ensure both files are in the same package; this will make it easier to use RoundButton.fx inside Chapter11.fx.

```
/*
 * RoundButton.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/23/2010 - created
 *
 * A Round Button using a SwingButton
 *
 */
```

```
package com.jfdimarzio.javafxforbeginners;

/***
 * @author JFDiMarzio
 */
```

You will be using both these files in the following section.

## Creating a RoundButton

In this section you create a new node, called RoundButton, that extends the SwingButton and overrides it to create a round button. You will create a round button that has two “modes.” In one mode, the text of the button is a plus sign (+) and in the other mode the text is a minus sign (-).

To begin, open the RoundButton.fx file you created earlier. Create a RoundButton that extends SwingButton, as follows:

```
public class RoundButton extends javafx.ext.swing.SwingButton{  
}
```

If you were to just stop here, you would have a fully functional SwingButton named RoundButton. You would be able to use this button in your script as you would any SwingButton.

This button works with so little code because you are telling it to extend the functionality of the SwingButton into RoundButton. Therefore, all the methods and properties of SwingButton will now be a part of RoundButton. In its current state, the round button looks and acts exactly like SwingButton.

However, for this example you want to create a button that is round. Therefore, you need to add code to the RoundButton class that will make the button round. In Chapter 10 you used the clip attribute to make a button round. You can apply the same technique here by overriding the clip attribute of SwingButton in RoundButton to always make the button round. Here’s the code:

```
javafx.scene.shape.Circle;  
  
override var clip = Circle {  
    centerX: 18,  
    centerY: 12  
    radius: 8  
};
```

This code serves the same function as it did in Chapter 10. You are still using a circle as a clip region to make this button round. The difference is that, by overriding the clip attribute to make the button round, you don’t need code when you use button in your script. No matter how many times you use the button, or where you put it, it will always be round.

Next, you are going to create an entirely new attribute named buttonType. This new attribute would only appear in RoundButton. The purpose of this new attribute is to let the

user choose whether the button text should be + or -. A buttonType of 1 would set the text of RoundButton to +, whereas a buttonType of 0 would set the text of RoundButton to -.

Set up a var named type. This var holds the value of buttonType when the user sets it:

```
var type:String = '-';
```

The buttonType attribute calls the type var and sets it accordingly:

```
public var buttonType:Number on replace{
    if (buttonType == 1) {
        type = '+';
    }else{
        type = '-';
    }
};
```

Let's take a look at this code. The code sets up a new public var named buttonType. Because the user will be setting this attribute to either 1 or 0, buttonType is typed as a Number. Although it is possible to have typed it as a Boolean, such a declaration would not make sense in this case. If the var were named isButtonTypePlusSign, that would be more useful as a Boolean. As it stands now, the buttonType should be a Number.

When the user does set buttonType, you want set the var *type* to either + or -. However, by default, a var cannot execute code. Variables only hold values; they do not execute code. You can force the attribute to execute code when its value is set by using the “on replace” trigger.

### NOTE

The “on replace” trigger can hold a block of code that runs when the value in the associated attribute is set.

Within the block of code that the on replace trigger will execute, you can use an if...else statement. The if...else statement tests an expression for a true or false outcome. If the expression evaluates as true, one block of code will execute; if it does not, a different block will execute.

In this case, you are checking the value of buttonType in the if statement. If buttonType is 1, the expression evaluates as true and type is set to +. If buttonType is 0, the expression evaluates to false and type is set to -.

The final step in creating the RoundButton is to set the text attribute. Simply override the text attribute and bind it to type. This will ensure that whenever buttonType is set, the text attribute will change accordingly. When buttonType is set to 1, the text of the button

will automatically be set to +. When buttonType is set to 0, the text of the button will automatically be set to -.

The finished script should look like this:

```
/*
 * RoundButton.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/23/2010 - created
 *
 * A Round Button using a SwingButton
 */
package com.jfdimarzio.javafxforbeginners;

import javafx.scene.shape.Circle;

/**
 * @author JFDiMarzio
 */

public class RoundButton extends javafx.ext.swing.SwingButton{
    var type:String = '-';

    override var clip = Circle {
        centerX: 18,
        centerY: 12
        radius: 8
    };

    public var buttonType:Number on replace{
        if (buttonType == 1){
            type = '+';
        }else{
            type = '-';
        }
    };
    override var text = bind type;
}
```

Unlike the other examples you have built in this book, you cannot compile and run this script as it is. What you have right now is a class that can be called from another script. This is the reason you created two empty script files at the beginning of this chapter.

Let's add some code to the second file, Chapter11.fx, to call and use the new RoundButton you've made.

Create a Stage and Scene in the Chapter11 script file as follows:

```
import javafx.stage.Stage;
import javafx.scene.Scene;

Stage {
    title : "Override Node"
    onClose: function () { }
    scene: Scene {
        width: 200
        height: 200
        content: [ ]
    }
}
```

Now all you have to do is call your new RoundButton from the content of the Scene. This is very easy to do. You can call RoundButton just as you would any other node. In this example, call RoundButton and set buttonType to 1:

```
RoundButton{
    buttonType:1;
    layoutX:0
    layoutY:0
}
```

This will create a round button with + as the text. The full script looks like this:

```
/*
 * Chapter11.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/23/2010 - created
 *
 * Custom Nodes and Overriding
 *
 */

package com.jfdimizaro.javafxforbeginners;
import javafx.stage.Stage;
import javafx.scene.Scene;

/***
 * @author JFDiMarzio
 */
```

```
Stage {
    title : "Override Node"
    onClose: function () { }
    scene: Scene {
        width: 200
        height: 200
        content: [RoundButton{
            buttonType:1;
            layoutX:0
            layoutY:0
        }]
    }
}
```

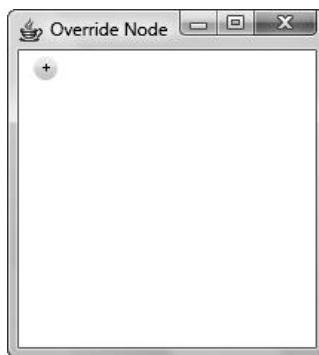
### NOTE

Some nodes, this RoundButton included, render differently on Mac and PC. Take this into consideration when comparing your results with those in the images in this chapter.

Compile and run this script. You will see a round button like the one shown in Figure 11-1.

To see one of the advantages of overriding an existing node to create your own, add a second button with a buttonType of 0, as follows:

```
/*
 * Chapter11.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/23/2010 - created
 */
```



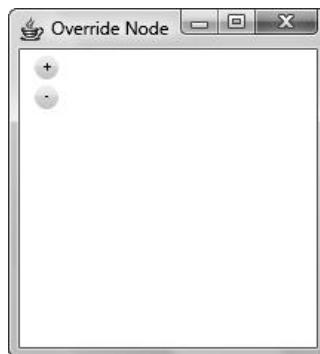
**Figure 11-1** A RoundButton

```
* Custom Nodes and Overriding
*
*/
package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;

/**
 * @author JFDiMarzio
 */
Stage {
    title : "Override Node"
    onClose: function () { }
    scene: Scene {
        width: 200
        height: 200
        content: [RoundButton{
            buttonType:1;
            layoutX:0
            layoutY:0
        }
        RoundButton{
            buttonType:0;
            layoutX:0;
            layoutY:20;
        }
    ]
}
}
```

Compile and run this script to produce the two buttons shown in Figure 11-2.



**Figure 11-2** A second RoundButton

It becomes very easy now to add multiple instances of nodes. If you know you will be adding multiple instances of a node, it is best to use this technique.

But what if you want to add a node that is unlike anything currently offered to you in JavaFX? For this you need to create your own custom node.

## Creating a Custom Node

What happens if you need a node that does not exist yet? That is, what if you want to do something in your application that would be much easier with a node that, unfortunately, is not available in JavaFX. This is a very real situation you will most likely run into over the course of your time programming in any language—JavaFX included.

When these situations arise, your best bet is to create your own node. JavaFX gives you an easy method for creating nodes that otherwise do not exist. You can create a node that will do exactly what you need it to do. Thus, you have complete control over this new node—what it will do, what it will look like, and how it will be used.

In the last section you created a class that extended SwingButton. In doing so, your new class inherited all the attributes and methods of SwingButton. However, now you want to make a node that is not only a button. Therefore, you can create a class that extends CustomNode, which will give you a clean node to begin building your class.

In this section you are going to create a new custom node that allows you to take notes. This node will have a text box for entering a note, a round button for adding the note, and a text area for displaying all the notes.

To begin, create a new, empty JavaFX script, as follows:

```
/*
 * Notes.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/23/2010 - created
 *
 * A Custom Node that uses RoundButton
 * - SwingTextField, and Text
 *
 */

package com.jfdimarzio.javafxforbeginners;

/**
 * @author JFDiMarzio
 */
```

```
public class Notes extends CustomNode{  
}
```

Notice that the Notes class extends CustomNode. This gives you a blank node to begin building on. Now let's create a list of what we need to add to the Notes class to create a note-taking node:

- A variable to hold the note text
- A SwingTextField to enter the note text
- A RoundButton to add the note
- A Text node to display all the notes taken

The first item on the list is to create a variable to hold the note text. This will give the Text node something to bind to when displaying a list of notes. To begin, create a var named notetext:

```
public class Notes extends CustomNode{  
    var notetext: String;  
}
```

Now, create a SwingTextField that you will use to enter your notes:

```
javafx.scene.CustomNode;  
  
public class Notes extends CustomNode{  
    var notetext: String;  
    var newNote : SwingTextField =  SwingTextField {  
        columns: 10  
        text: "Add new note"  
        editable: true  
    }  
}
```

So far this has been pretty straightforward. However, now you need to tackle some work that has not been covered yet. How do you display this SwingTextField? CustomNodes do not hold Scenes and Stages in the way you are used to working with them. Rather, nodes are created and used in scripts.

All nodes have a create method. This method returns a node or group of nodes to whatever is calling (or creating) the node. For example, if you use a Text node in a Scene, the Scene calls the create method of the Text node. The create method does whatever work is needed to create the Text node and returns the created Text node to the Scene.

Therefore, to create your new Notes node, you will need to override the CustomNode's create method to return your new group of nodes:

```
javafx.ext.swing.SwingTextField;

public class Notes extends CustomNode{
    var notetext: String;
    var newNote : SwingTextField =  SwingTextField {
        columns: 10
        text: "Add new note"
        editable: true
    }

    override function create():Node{
}
}
```

### **NOTE**

Notice that the create() method is typed as :Node. This tells you that the result of create returns a node.

You will be returning a group of nodes through the create function because you are displaying more than one node to the screen. In fact, you are displaying three different nodes. Let's start building the create method.

The create method needs to build all three nodes as well as supply the action for the RoundButton to add notes. This is a fairly easy process. First, create the Node group:

```
javafx.scene.Group;

override function create():Node{
    return Group{
        content: [
        ]
    }
}
```

Notice that the group contains a content attribute. All the nodes you want to return will go within this content. Start by adding your newNote SwingTextField:

```
override function create():Node{
    return Group{
        content: [
            newNote,
        ]
    }
}
```

Now, let's create a RoundButton. This is the same RoundButton you created in the last section of this chapter. Because this button is used for adding notes, set the buttonType to 1; this will create a button with + as the text. Additionally, you need to create an anonymous function for the button's action. The purpose of this function is to take the text of the SwingTextField and add it to the notetext variable being displayed:

```
override function create():Node{
    return Group{
        content: [
            newNote,
            RoundButton{
                layoutX: 150;
                layoutY: 0;
                buttonType:1;
                action: function(){
                    notetext = "{notetext} \n {newNote.text}";
                }
            }
        ]
    }
}
```

The only difference you see here with the implementation of the RoundButton is the addition of the action anonymous function. This function sets notetext to the current contents of notetext plus a newline character (\n) and the current text of the newNote text field. This displays each note entered on a new line, like this:

```
Note #1
Note #2
Note #3
```

Now, all you have to do is create a new Text node to display notetext. Simply create a Text node whose context is bound to notetext:

```
Text{
    x: 0;
    y: 35;
    content: bind notetext;
}
```

Binding the content of the Text node will ensure that when notetext is updated by the RoundButton, the latest updates are displayed directly to the Text node. So with very

little effort you have created a new node that takes and displays notes. The final script for Notes.fx should appear as follows:

```
/*
 * Notes.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/23/2010 - created
 *
 * A Custom Node that uses RoundButton
 * - SwingTextField, and Text
 *
 */

package com.jfdimarzio.javafxforbeginners;

import javafx.scene.CustomNode;
import javafx.scene.text.Text;
import javafx.scene.Node;
import javafx.scene.Group;
import javafx.ext.swing.SwingTextField;

/**
 * @author JFDiMarzio
 */

public class Notes extends CustomNode{
    var notetext: String;
    var newNote : SwingTextField =  SwingTextField {
        columns: 10
        text: "Add new note"
        editable: true
    }

    override function create():Node{
        return Group{
            content:[
                newNote,
                RoundButton{
                    layoutX: 150;
                    layoutY: 0;
                    buttonType:1;
                    action: function(){
                        notetext = "{notetext} \n  {newNote.text}";
                    }
                }
            ]
        }
    }
}
```

```
        Text{  
            x: 0;  
            y: 35;  
            content: bind notetext;  
        }  
    ]  
}  
}  
}
```

With the Notes.fx script completed, it is time to use it. Edit the Chapter11.fx file you used in the last section to display the Notes class rather than RoundButton. Your new Chapter11.fx script should look like this:

```
/*  
 * Chapter11.fx  
 *  
 * v1.0 - J. F. DiMarzio  
 *  
 * 6/23/2010 - created  
 *  
 * Custom Nodes and Overriding  
 *  
 */  
  
package com.jfdimarzio.javafxforbeginners;  
  
import javafx.stage.Stage;  
import javafx.scene.Scene;  
  
/**  
 * @author JFDiMarzio  
 */  
  
Stage {  
    title: "Override Node"  
    onClose: function () {  
    }  
    scene: Scene {  
        width: 200  
        height: 200  
        content: [Notes {  
        }  
    ]  
}  
}
```

Compile and run this script. You should see your custom node, as shown in Figure 11-3.

```
/*
 * Chapter11.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/23/2010 - created
 *
 * Custom Nodes and Overriding
 *
 */
package com.jfdimarzio.javafxforbeginners;

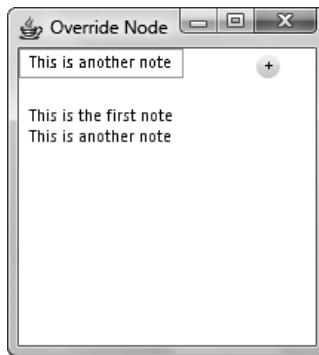
import javafx.stage.Stage;
import javafx.scene.Scene;

/**
 * @author JFDiMarzio
 */
Stage {
    title: "Override Node"
    onClose: function () {
    }
    scene: Scene {
        width: 200
        height: 200
        content: [Notes {
            }
        ]
    }
}
```



---

**Figure 11-3** The custom Notes node



**Figure 11-4** The Notes node with notes entered

This custom node is set up with a text field for you to enter notes in and a button to add the notes. Enter a few notes, hitting the + button after each note. You should see your notes stack up as shown in Figure 11-4.

## Try This Create Your Own Shapes

In this chapter you used a mask to create a round button. For an added challenge, try this exercise to expand your knowledge. Create a new project and, using the skills you gained from this chapter, create different shapes of different nodes. Experiment with triangular buttons, rounded-corner text boxes, and other nontraditional shapes. Then, using the RoundButton example, create a custom node of your new shape and use it in a Scene.

In the next chapter you will learn how to embed music and videos in your scripts.



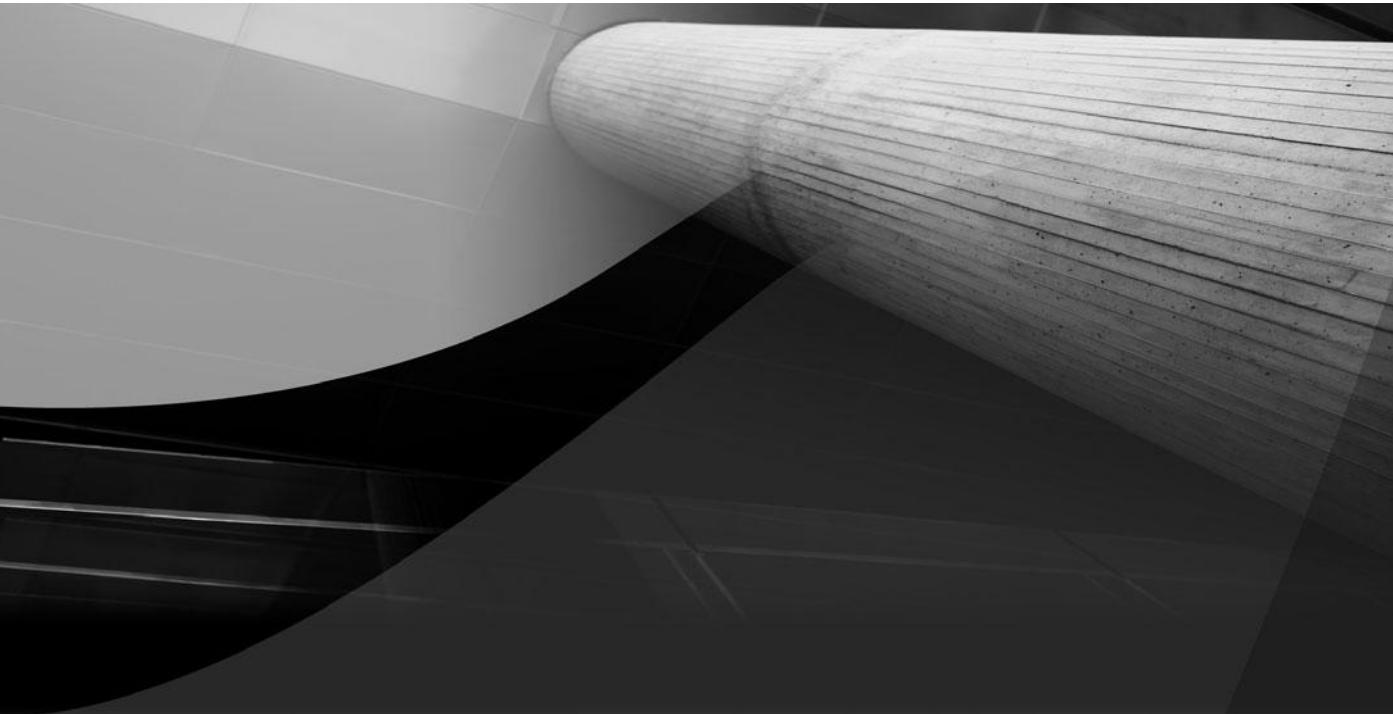
## Chapter 11 Self Test

1. What process lets you take methods and properties from one class and change their default actions and behaviors?
2. When you're creating a class, what keyword forces your class to inherit the methods and properties of another?

3. In the following example, what will a call to YourDog.displayBreed print?

```
public class MyDog extends Dog{  
    override function displayBreed(){  
        println("Elkhound");  
    }  
}  
public class YourDog extends Dog{  
}
```

4. True or false? Ensuring that your files are all in the same package will make referencing them easier.
5. True or false? After inheriting from a class, only attributes you override are available to you in another class.
6. What trigger will execute when an attribute changes?
7. What statement will take an expression for a true or false result and then execute code accordingly?
8. True or false? You have to call a custom-created node from a script to use it.
9. What node do you inherit from to create a custom node?
10. What method of CustomNode do you override to return your node to the calling script?



# Chapter 12

Embedded Video  
and Music

## Key Skills & Concepts

- Adding video to an app
  - Inverse binding
  - Using the MediaPlayer
- 

With the abundance of business and sites built around streaming media, it is only natural that JavaFX allow you to take full advantage of video and audio media. In the past, programming an application to play any kind of media meant hours of writing controls, finding the right codecs for the files you wanted to play, and writing parsers to read the media files. It was long and arduous work that you would really only attempt if you were completely confident in your abilities as a programmer.

JavaFX has packaged a few nodes that make working with media files as easy as possible. In this chapter you learn about the MediaView and MediaPlayer nodes. You use these nodes to write apps that display video, play audio, and allow users a level of control over the media they choose to play.

In the first section, you learn how MediaView and MediaPlayer can be used together to play a video file. However, before you begin, you need to create a new, empty JavaFX script. Your script should appear as follows:

```
/*
 * Chapter12.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/25/2010 - created
 *
 * Embedded Video and Music
 *
 */
package com.jfdimarzio.javafxforbeginners;

/**
 * @author JFDiMarzio
 */
```

You also need to create a second empty JavaFX script. This second file will hold a custom node for the playback of media files. You will be adding to this second file throughout the chapter. Name the file **MyMediaPlayer.fx**, and set it up as follows:

```
/*
 * MyMediaPlayer.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/25/2010 - created
 *
 * Embedded Video and Music
 *
 */
```

```
package com.jfdimarzio.javafxforbeginners;
```

```
/***
 * @author JFDiMarzio
 */
```

In the following sections you learn how easy it is to add video files and audio files to your applications.

## Playing Video

JavaFX uses the MediaView node to display video to the screen. However, the MediaView is really just a container for a MediaPlayer. The MediaPlayer node is the one that plays the video.

In this section you use a MediaView and a MediaPlayer to play some sample video from within your application. You will be surprised at just how easy JavaFX makes it to play video from a script.

### TIP

The sample video used in this section can be downloaded from <http://www.microsoft.com/windows/windowsmedia/musicandvideo/hdvideo/contentshowcase.aspx>.

Let's start by creating a MediaPlayer. After the MediaPlayer is created, you will add it to a MediaView in your Chapter12 script.

To begin, open your MyMediaPlayer.fx script. The nodes needed to work with media files are located in the import javafx.scene.media package. JavaFX does a really good job of naming everything logically and placing all related classes in common packages. The javafx.scene.media package is no exception.

Import the following package to your MyMediaPlayer file:

```
import javafx.scene.media.MediaPlayer;
```

Now, establish a new MyMediaPlayer class that extends MediaPlayer:

```
public class MyMediaPlayer extends MediaPlayer {  
}
```

So why go through the extra step of creating a custom node for the MediaPlayer? Although it is true that you could just add the MediaPlayer directly to a MediaView and be done with it, the stock MediaPlayer is very simplistic. As you progress through this chapter you will be adding functionality to MyMediaPlayer that extends the usability of MediaPlayer.

For now you override two properties of the MediaPlayer, just to show how easy it is to load and play a video file. The autoPlay property tells the MediaPlayer to start playing the file immediately, as soon as it is loaded. Let's override the autoPlay property to be true:

```
public class MyMediaPlayer extends MediaPlayer {  
    override var autoPlay = true;  
}
```

The second override we want to make is to the media property. The media property is a Media node and represents the actual media file to be played using the MediaPlayer. For this example you are going to override the media property to hard-code a sample video file. This saves you the effort of constantly having to assign this property.

## Ask the Expert

**Q: What kind of files can the MediaPlayer play?**

**A:** The quick answer is, if the file is supported by QuickTime or Windows Media Player, the MediaPlayer node will be able to handle it. Specific supported file formats include MP3, MP4, WMV, FLV, 3GP, and MOV.

Import the following statement to establish a Media node:

```
import javafx.scene.media.Media;
```

The code to override the media property of the MediaPlayer should look like this:

```
public class MyMediaPlayer extends MediaPlayer {  
    override var autoPlay = true;  
    override var media = Media {  
        source: "file:///C:/wmdownloads/Robotica_720.wmv"  
    }  
}
```

In this example, the custom MediaPlayer you have created always plays the Robotica\_720.wmv file from the wmdownloads folder of the local drive. The finished MyMediaPlayer script should appear as follows:

```
/*  
 * MyMediaPlayer.fx  
 *  
 * v1.0 - J. F. DiMarzio  
 *  
 * 6/25/2010 - created  
 *  
 * Embedded Video and Music  
 *  
 */  
  
package com.jfdimarzio.javafxforbeginners;  
  
import javafx.scene.media.MediaPlayer;  
import javafx.scene.media.Media;  
  
/**  
 * @author JFDiMarzio  
 */  
  
public class MyMediaPlayer extends MediaPlayer {  
    override var autoPlay = true;  
    override var media = Media {  
        source: "C:/wmdownloads/Robotica_720.wmv"  
    }  
}
```

Now you can use MyMediaPlayer just as you would use the standard MediaPlayer node. Let's add a MediaView to the Chapter12.fx script and assign MyMediaPlayer to it.

Given that you have not yet added any code to the Chapter12.fx script, you need to import the correct package for the MediaView, as follows:

```
import javafx.scene.media.MediaView;

var myMediaPlayer : MyMediaPlayer = MyMediaPlayer {  
}
```

The only property you need to set right now in your MediaView is mediaPlayer. The mediaPlayer property holds your MediaPlayer node, and as you discovered earlier in this section, the MediaPlayer node plays your media file. Here, you have created a var named myMediaPlayer of your custom MediaPlayer. This var can be assigned to the mediaPlayer property of your MediaView, as shown here:

```
MediaView {  
    mediaPlayer : myMediaPlayer  
}
```

This code is pretty self-explanatory. You are creating a MediaView node to hold your MediaPlayer and send the MediaPlayer to the screen. MyMediaPlayer (the MediaPlayer you created earlier in the section) is assigned to the mediaPlayer property of the MediaView.

The full Chapter12.fx script file should appear as follows:

```
/*
 * Chapter12.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/25/2010 - created
 *
 * Embedded Video and Music
 *
 */

package com.jfdimizaro.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.media.MediaView;

var myMediaPlayer : MyMediaPlayer = MyMediaPlayer {  
}

Stage {  
    title : "EmbeddedMedia"  
    onClose: function () {    }  
}
```

```
scene: Scene {
    width: 1280
    height: 720
    content: [ MediaView {
        mediaPlayer : myMediaPlayer
    }
]
}
```

Compile and run this example. You will see a sample video begin to play as soon as the application is loaded. The file plays automatically because you overrode the autoPlay property to be true.

If you look a little more closely at the video playback, you may notice that it is very simplistic: There are no controls for playing and pausing the video. There are also no controls to indicate the progress of the video or to allow the user to skip through the video at will. In short, there are no controls whatsoever in the script.

To give your users control over the playback of a video, you need to write the controls yourself. In the remainder of this section, you write some controls on your MediaPlayer and present them to the user.

### TIP

Because you thought ahead and already created a custom MediaPlayer, the process for creating your controls will be much easier.

Let's create a quick button to control the playback of the video.

## Creating a Play/Pause Button

In this section you create a play/pause button using the RoundButton custom button you created earlier in this book. If you do not have all the code for the RoundButton readily available, don't worry. The full RoundButton script should look like this (you can re-create it from here if needed):

```
/*
 * RoundButton.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/23/2010 - created
 *
 * A Round Button using a SwingButton
 */
```

```
package com.jfdimarzio.javafxforbeginners;

import javafx.scene.shape.Circle;

/**
 * @author JFDiMarzio
 */

public class RoundButton extends javafx.ext.swing.SwingButton{
    var type:String = '||';

    override var clip = Circle {
        centerX: 18,
        centerY: 12
        radius: 8
    };

    public var buttonType:Number on replace{
        if (buttonType == 1){
            type = '>';
        }else{
            type = '||';
        }
    };

    override var text = bind type;
}
```

One cosmetic change has been made to this version of the RoundButton. The type variable in this version of RoundButton has been changed from + or – to > or ||. This indicates whether the button is being used for playing the video or pausing the video, respectively.

### TIP

If you need a refresher for how the RoundButton works, refer to Chapter 11.

Next, let's change one setting in your MyMediaPlayer script. Change the autoPlay property from true to false, as shown next. This keeps the video from playing until you click the play button.

```
/*
 * MyMediaPlayer.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/25/2010 - created
 *
```

```
* Embedded Video and Music
*
*/
package com.jfdimarzio.javafxforbeginners;

import javafx.scene.media.MediaPlayer;
import javafx.scene.media.Media;

/**
 * @author JFDiMarzio
 */
public class MyMediaPlayer extends MediaPlayer {
    override var autoPlay = false;
    override var media = Media {
        source: "C:/wmdownloads/Robotica_720.wmv"
    }
}
```

Now that the button is ready and the MediaPlayer is ready, the last step is to set up the Chapter12.fx script to use the button in controlling the video.

The first step is to establish a variable that can be used to track what mode the video is currently in (play or pause):

```
var playVideo : Integer = 1;
```

You can bind your button to this value to determine if the video is currently being played or is currently paused. The button will really do all the work here, but you do need to help it along a little. You need to write a function for the button's action that will play or pause the video based on the bound value of playVideo.

Use an if...else statement to test the value of playVideo; then call either myMediaPlayer.play() or myMediaPlayer.pause() accordingly. The function within your button's action should look like this:

```
if (playVideo == 1){
    myMediaPlayer.play();
    playVideo = 0
} else{
    myMediaPlayer.pause();
    playVideo = 1
}
```

Notice that if the var playVideo is 1, the video is played and playVideo is set to 0. By setting playVideo to 0, you ensure that the next time the button is clicked, the video will be paused.

The full code of the Chapter12.fx script should appear as follows:

```
/*
 * Chapter12.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/25/2010 - created
 *
 * Embedded Video and Music
 *
 */

package com.jfdimizaro.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.media.MediaView;

var myMediaPlayer: MyMediaPlayer = MyMediaPlayer {
    }
var playVideo: Integer = 1;

Stage {
    title: "EmbeddedMedia"
    onClose: function () {
    }
    scene: Scene {
        width: 1280
        height: 720
        content: [MediaView {
            mediaPlayer: myMediaPlayer
        }]
        RoundButton {
            buttonType: bind playVideo;
            action: function () {
                if (playVideo == 1) {
                    myMediaPlayer.play();
                    playVideo = 0
                } else {
                    myMediaPlayer.pause();
                    playVideo = 1
                }
            }
        }
    }
}
```

Compile and run this example. Your video will now load, but it will not play until you click the > button. Once the video plays, the button displays “||” and clicking it will cause the video to pause.

Now that you have created a simple play/pause button, let’s make a progress indicator that allows the user to advance and rewind the video.

## Creating a Progress Indicator

To create a progress indicator, you need to know two key pieces of information. First, you need to know the total length of the video you are playing. Second, you need to know at what point in time of the playback you are currently. Luckily, JavaFX can tell you both of these things. However, it may take some massaging of the data to get it into a usable format.

Within the MyMediaPlayer control is a media property. The media property is the Media node of the actual video file being played. The Media node has a property called duration. Therefore, to get the total running time of the video being played, you can make the following call:

```
myMediaPlayer.media.duration.toMillis();
```

The toMillis() method of duration translates the duration into milliseconds. Milliseconds are much easier to work with and are much more flexible than a Duration object.

To get the current playback time, you need to be a little more creative. The MediaPlayer node has a property called currentTime, which also returns a Duration object that represents the current time position of the playback. Why do we need to get creative if the MediaPlayer already has a property for the current time? Keep in mind that you are creating a control that not only marks off the current playback but allows you to set the current playback position as well. For this you need to work in something a little more flexible than a Duration object.

The control you will be using as your progress indicator is a Slider. The position of the Slider’s toggle is controlled by a value property, which is typed as a Number. Therefore, it would be much easier for you if you could convert the MediaPlayer’s currentTime into a Number; this would allow you to bind the currentTime directly to the Slider.

Because currentTime is already typed as a Duration, you cannot change it to a Number, even if you override it (you cannot change the type of a property during an override). Therefore, you have to create a new property in MyMediaPlayer that can hold a translated version of the currentTime property.

Create a new public var in MyMediaPlayer called **progressIndicator**. In this property you are going to set currentTime to the Duration of progressIndicator, as follows:

```
public var progressIndicator: Number = 0 on replace {  
    if (media.duration.toMillis() != java.lang.Double.NEGATIVE_INFINITY  
        and media.duration.toMillis() != java.lang.Double.POSITIVE_INFINITY) {  
        currentTime = media.duration.valueOf(progressIndicator);  
    }  
};
```

In this property, after some checks to make sure that the duration of the video is valid, you use duration.valueOf() to convert progressIndicator into a currentTime value. Now, when progressIndicator is set to a value (in milliseconds), it will move the video to that time by setting currentTime. The next step is to set progressIndicator to the currentTime value, thus giving you something to bind to.

Override the currentTime property as follows:

```
override var currentTime on replace {  
    if(media.duration.toMillis() != java.lang.Double.NEGATIVE_INFINITY  
        and  
        media.duration.toMillis() != java.lang.Double.POSITIVE_INFINITY  
        and  
        media.duration.toMillis() != 0 and  
        currentTime.toMillis() != java.lang.Double.NEGATIVE_INFINITY  
        and  
        currentTime.toMillis() != java.lang.Double.POSITIVE_INFINITY) {  
        progressIndicator = currentTime.toMillis()  
    }  
}
```

Much like the progressIndicator property, the override for currentTime first checks that the media duration is valid and then that the currentTime is valid. Assuming it passes these checks, progressIndicator is set to currentTime converted into milliseconds.

You now have the values needed to get and set the current playback time of the video. The full MyMediaPlayer file should now look like this:

```
/*  
 * MyMediaPlayer.fx  
 *  
 * v1.0 - J. F. DiMarzio  
 *  
 * 6/25/2010 - created  
 *  
 * Embedded Video and Music  
 *  
 */  
package com.jfdimarzio.javafxforbeginners;
```

```
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.Media;

/**
 * @author JFDiMarzio
 */
public class MyMediaPlayer extends MediaPlayer {

    override var autoPlay = false;
    public var progressIndicator: Number = 0 on replace {
        if (media.duration.toMillis() != java.lang.Double.NEGATIVE_INFINITY
            and media.duration.toMillis() != java.lang.Double.POSITIVE_INFINITY) {
            currentTime = media.duration.valueOf(progressIndicator);
        }
    };
    override var currentTime on replace {
        if(media.duration.toMillis() != java.lang.Double.NEGATIVE_INFINITY
            and
            media.duration.toMillis() != java.lang.Double.POSITIVE_INFINITY
            and
            media.duration.toMillis() != 0 and
            currentTime.toMillis() != java.lang.Double.NEGATIVE_INFINITY
            and
            currentTime.toMillis() != java.lang.Double.POSITIVE_INFINITY) {
            progressIndicator = currentTime.toMillis()
        }
    }
    override var media = Media {
        source: "C:/wmdownloads/Robotica_720.wmv"
    }
}
```

With the values in MyMediaPlayer created and ready to be used, you can now set up your Slider.

In earlier chapters of this book, you learned how to use bind. Binding allows you to set values in properties that can be updated dynamically. The Slider you are building needs to be set dynamically to the current playback time of the video using bind. However, the Slider also needs to set the current playback time if it is changed. To accomplish this you need to use a new binding concept:

```
bind...with inverse
```

The with inverse modifier of bind allows the binding to work in both directions. Thus, the Slider value can be set from the current playback time, and changing the Slider value will also change the current playback time.

There is one caveat to using inverse binding: You cannot bind directly to the value that you want to use. Rather, you must indirectly bind to it through a variable. This concept will become clearer as you begin coding.

First, in the Chapter12.fx script, add a variable named **progress**, typed as a Number:

```
var progress : Number ;
```

Next, in the **myMediaPlayer** variable you created in the Chapter12.fx script, inverse-bind **progressIndicator** to the new **progress** variable:

```
var myMediaPlayer: MyMediaPlayer = MyMediaPlayer {  
    progressIndicator: bind progress with inverse;  
}
```

Binding **progressIndicator** inversely to **progress** allows a bidirectional update of **myMediaPlayer.progressIndicator** through **progress**.

The final step is to create a **Slider** and inversely bind the **value** property to **progress**, as follows:

```
import javafx.scene.control.Slider;  
Slider {  
    translateX: 35  
    translateY: 5  
    min: 0  
    max: bind myMediaPlayer.media.duration.toMillis()  
    value: bind progress with inverse;  
    vertical: false  
}
```

The final Chapter12.fx script should appear as follows:

```
/*  
 * Chapter12.fx  
 *  
 * v1.0 - J. F. DiMarzio  
 *  
 * 6/25/2010 - created  
 *  
 * Embedded Video and Music  
 *  
 */  
package com.jfdimarzio.javafxforbeginners;  
  
import javafx.stage.Stage;  
import javafx.scene.Scene;
```

```
import javafx.scene.media.MediaView;
import javafx.scene.control.Slider;

var myMediaPlayer: MyMediaPlayer = MyMediaPlayer {
    progressIndicator: bind progress with inverse;
}

var playVideo: Integer = 1;

var progress : Number ;

Stage {
    title: "EmbeddedMedia"
    onClose: function () {
    }
    scene: Scene {
        width: 1280
        height: 720
        content: [MediaView {
            mediaPlayer: myMediaPlayer
        }]
        RoundButton {
            buttonType: bind playVideo;
            action: function () {
                if (playVideo == 1) {
                    myMediaPlayer.play();
                    playVideo = 0
                } else {
                    myMediaPlayer.pause();
                    playVideo = 1
                }
            }
        }
        Slider {
            translateX: 35
            translateY: 5
            min: 0
            max: bind myMediaPlayer.media.duration.toMillis()
            value: bind progress with inverse;
            vertical: false
        }
    ]
}
}
```

Compile and run these scripts. You now have a pause/play button and a progress indicator that allows the user to change the current playback position of the video.

In the next section you learn how to use the MediaPlayer for audio playback.

## Playing Audio

Audio files such as MP3s can be played just as easily as video files. In fact, simply changing the media source to an MP3 file is all that is required to turn your video player into an audio player. The following change plays the MP3 “Dream Police” by Cheap Trick:

```
override var media = Media {  
    source: "C:/wmdownloads/CheapTrick-DreamPolice.mp3"  
}
```

### NOTE

As noted earlier in this chapter, there also seems to be a compatibility issue with the JavaFX media player and Mac. Hopefully this will be corrected in future releases.

JavaFX has made the playing of media files, of any format, a very easy task to take on. Using the skills you learned earlier in this chapter, create a vertical Slider to control the volume on your audio player. (Hint: The mediaPlayer has a property called volume that you can bind to as is.)

Compare your solution with the code that follows:

```
/*  
 * Chapter12.fx  
 *  
 * v1.0 - J. F. DiMarzio  
 *  
 * 6/25/2010 - created  
 *  
 * Embedded Video and Music  
 *  
 */  
  
package com.jfdimarzio.javafxforbeginners;  
  
import javafx.stage.Stage;  
import javafx.scene.Scene;  
import javafx.scene.media.MediaView;  
import javafx.scene.control.Slider;  
import javafx.scene.media.MediaError;  
  
var myMediaPlayer: MyMediaPlayer = MyMediaPlayer {  
    progressIndicator: bind progress with inverse;  
    volume: bind playbackVolume with inverse;  
}  
  
var playVideo: Integer = 1;
```

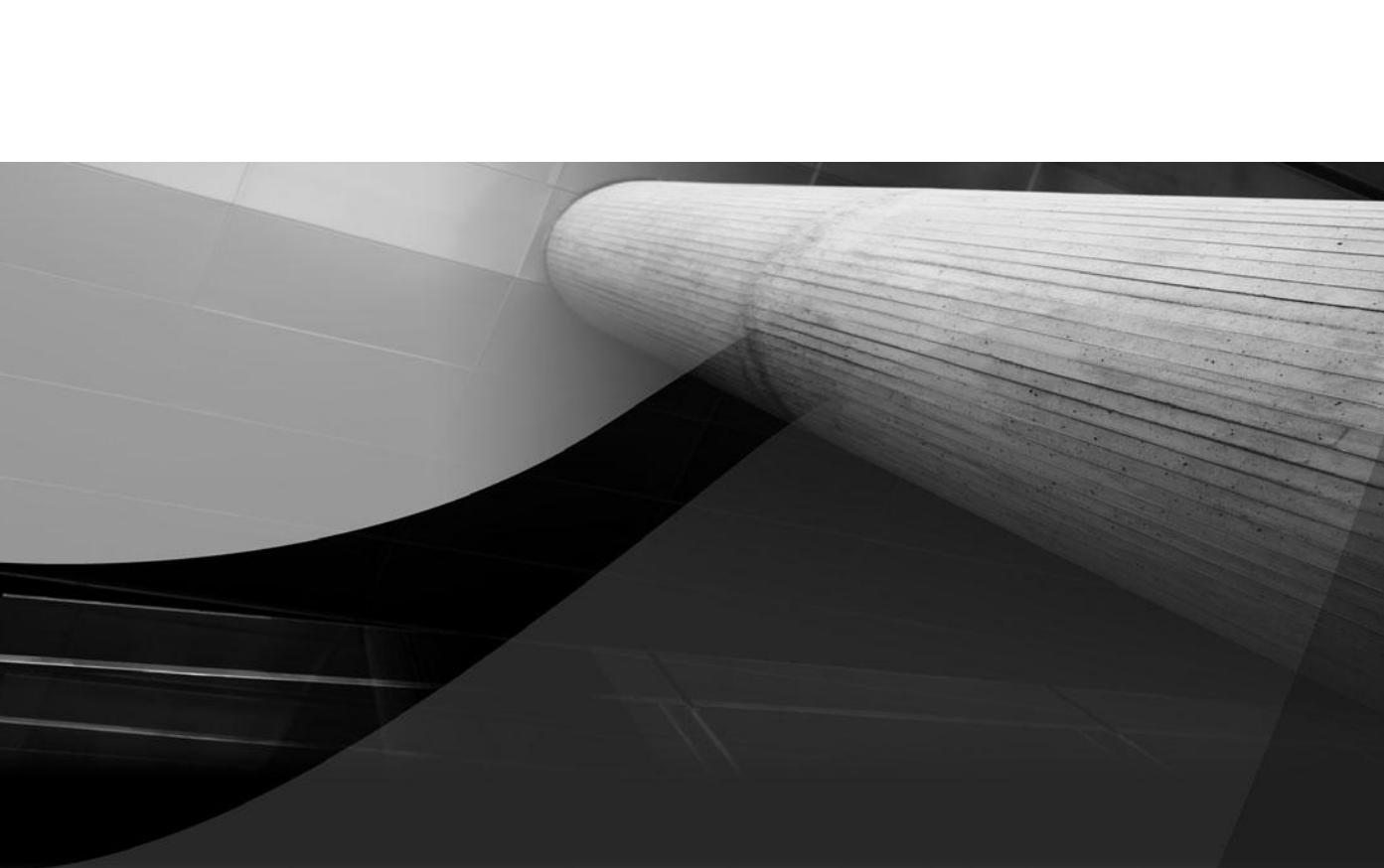
```
var progress : Number ;  
  
var playbackVolume : Number = 2 ;  
  
Stage {  
    title: "EmbeddedMedia"  
    onClose: function () {  
    }  
    scene: Scene {  
        width: 200  
        height: 200  
        content: [MediaView {  
            mediaPlayer: myMediaPlayer  
        }  
        RoundButton {  
            buttonType: bind playVideo;  
            action: function () {  
                if (playVideo == 1) {  
                    myMediaPlayer.play();  
                    playVideo = 0  
                } else {  
                    myMediaPlayer.pause();  
                    playVideo = 1  
                }  
            }  
        }  
        Slider {  
            translateX: 35  
            translateY: 5  
            min: 0  
            max: bind myMediaPlayer.media.duration.toMillis()  
            value: bind progress with inverse;  
            vertical: false  
        }  
        Slider {  
            translateX: 5  
            translateY: 35  
            min: 0  
            max: 5  
            value: bind playbackVolume with inverse;  
            vertical: true  
            rotate: 180  
        }  
    ]  
}
```

In the next chapter you learn how to use JavaFX layouts.



## Chapter 12 Self Test

1. What node is used to hold a MediaPlayer?
2. What package contains all the nodes needed to work with media files?
3. What property of the MediaPlayer tells the media file to play once it has loaded?
4. What media formats can the MediaPlayer play?
5. What property of the MediaPlayer will pause media playback?
6. True or false? MediaPlayer.mediaLength() will give you the total running time of a media file.
7. What type is MediaPlayer.currentTime?
8. What type of binding allows for bidirectional updating?
9. True or false? Using inverse binding, you can bind directly to a value.
10. What property of MediaPlayer can you bind to in controlling the playback volume?



# Chapter 13

## Using JavaFX Layouts

## Key Skills & Concepts

- Arranging nodes in a Scene
  - Using the HBox
  - Nested layouts
- 

Throughout this book you have learned how to create and use nodes. At its core, JavaFX would not be very interesting without the ability to place these nodes on the screen. Nodes need to be placed logically on the screen for any user interface (UI) to be intuitive and usable.

To this point in the book, you have manually placed nodes within the content of a Scene. Then, using the x and y coordinates of the desired position, you have moved those nodes around to put them in a logical place. However, manually moving nodes around the screen can be a tedious task and ultimately take away from the time needed to develop functionality.

JavaFX provides a handful of very useful tools to help you organize the placement of nodes on the screen. Layouts can automatically organize your nodes into predefined patterns that will give your applications a professional look and a higher degree of usability.

In this chapter you learn how to use layouts to organize the various nodes. The first layout you learn about is the HBox.

Before you begin, create a new, empty JavaFX script named **Chapter13.fx**. The script should look like this:

```
/*
 * Chapter13.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/23/2010 - created
 *
 * Using JavaFX Layouts
 *
 */
package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
```

```
import javafx.scene.layout.HBox;
import javafx.scene.control.TextBox;
import javafx.scene.control.Button;

/***
 * @author JFDiMarzio
 */
```

## The HBox

The HBox is a horizontal layout you can use to place relatively positioned nodes on the screen. That is, the HBox layout organizes your nodes next to each other horizontally. To demonstrate this, let's create a Scene with a TextBox and a Button, as follows:

```
TextBox {
    text: "SampleText"
    columns: 12
    selectOnFocus: true
} Button {
    text: "Button"
    action: function () {
    }
}
```

Your full script should now look like this:

```
/*
 * Chapter13.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/23/2010 - created
 *
 * Using JavaFX Layouts
 *
 */
package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.TextBox;
import javafx.scene.control.Button;

/***
 * @author JFDiMarzio
 */
```

```
Stage {
    title: "UsingLayouts"
    onClose: function () {
    }
    scene: Scene {
        width: 200
        height: 200
        content: [TextBox {
            text: "SampleText"
            columns: 12
            selectOnFocus: true
        } Button {
            text: "Button"
            action: function () {
            }
        }]
    }
}
```

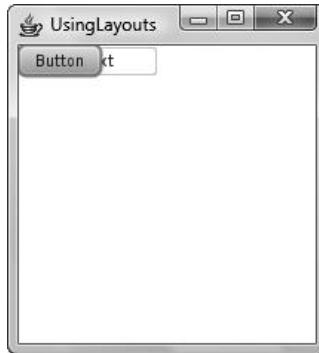
Compile and run this script. You can see that if you do not modify the layout of the nodes, they will simply appear jumbled together, as shown in Figure 13-1.

The layout of these nodes is very discordant and does not lend itself to a very usable design. Let's use the HBox layout to organize these nodes on the screen. The HBox belongs to the javafx.scene.layout package. You must import this package before you add the HBox to your Scene.

### TIP

Remove the TextBox and Button from the last example now. You will re-add them soon, but this time as the content of the HBox.

```
import javafx.scene.layout.HBox;
```



---

**Figure 13-1** Nodes without a layout

After you have imported the correct package, add the HBox to your Scene:

```
Stage {
    title: "UsingLayouts"
    onClose: function () {
    }
    scene: Scene {
        width: 200
        height: 200
        content: [HBox {
            content: []
        }]
    }
}
```

Right now, the only property of the HBox you will be concerned with is the content property. You add all the nodes you want the HBox to organize to the content property of the HBox. The HBox will place all the nodes horizontally within its content.

Add a TextBox and a Button to the content property of the HBox, as follows:

```
HBox {
    content: [TextBox {
        text: "SampleText"
        columns: 12
        selectOnFocus: true
    } Button {
        text: "Button"
        action: function () {
    }
}
]
```

Notice that you are still not setting the x and y coordinate positions of the TextBox and the Button. Under normal circumstances, this would result in the nodes being placed on top of each other. However, the HBox takes care of the placement for you and arranges the nodes neatly next to each other. The full code for your script should appear as follows:

```
/*
 * Chapter13.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/23/2010 - created
 *
 * Using JavaFX Layouts
 */
```

```
package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.control.TextBox;
import javafx.scene.control.Button;

/**
 * @author JFDiMarzio
 */
Stage {
    title: "UsingLayouts"
    onClose: function () {
    }
    scene: Scene {
        width: 200
        height: 200
        content: [HBox {
            content: [TextBox {
                text: "SampleText"
                columns: 12
                selectOnFocus: true
            } Button {
                text: "Button"
                action: function () {
                }
            }
        ]
    }
}
}
```

Compile this script and run it. Your node will appear as shown in Figure 13-2.

The HBox layout is simple to use and does not require any parameters to be set to give you great results. It is very easy to line up and organize your nodes using this tool.

Keep in mind that all the layouts in JavaFX, including the HBox, inherit from Node. This means that events such as `onKey*` and `onMouse*` can be used in layouts. Also, you can apply effects to layouts as well, thus giving you a large array of customization options. For more information about using events or effects in nodes, refer to the earlier chapters of this book.

In the next section you use the VBox to organize your nodes vertically rather than horizontally.



**Figure 13-2** Nodes placed using an HBox

## The VBox

The VBox is a JavaFX layout that organizes your nodes vertically rather than horizontally like the HBox does. The VBox can be particularly useful in creating forms and other applications where information is generally used in a top-down manner.

Import the `javafx.scene.layout.VBox` package to use the VBox layout:

```
import javafx.scene.layout.VBox;
```

The VBox is easy to implement. Add a VBox control to your Scene as shown in the following code sample:

```
Scene {
    width: 200
    height: 200
    content: [VBox {
        content: [
        ]
    }
]
```

Using the same TextBox and Button you used in the previous section's example, let's create a VBox layout. The finished script should look like this:

```
/*
 * Chapter13.fx
 *
 * v1.0 - J. F. DiMarzio
 *
```

```
* 6/23/2010 - created
*
* Using JavaFX Layouts
*
*/
package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.control.TextBox;
import javafx.scene.control.Button;

/**
 * @author JFDiMarzio
 */
Stage {
    title: "UsingLayouts"
    onClose: function () {
    }
    scene: Scene {
        width: 200
        height: 200
        content: [VBox {
            content: [TextBox {
                text: "SampleText"
                columns: 12
                selectOnFocus: true
            } Button {
                text: "Button"
                action: function () {
                }
            }
        ]]
    }
}
```

Compile and run this script. In the previous section, you used the HBox to organize nodes in a horizontal line. Using the VBox in this script, you can see that the same two nodes now fall under each other vertically. Figure 13-3 shows this result.

Again, this example shows just how easy it is to use a layout in JavaFX to organize your nodes. In the next section you combine what you have learned so far about layouts to create a nested layout.

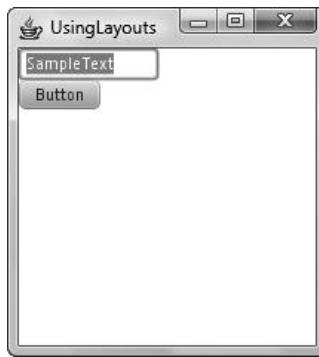


Figure 13-3 Organizing nodes using VBox

## Nested Layouts

One of the more flexible aspects of layouts is that they can be nested. That is to say, you can place layouts within each other. You can create some very useful UIs by combining two or more existing layouts.

In the previous sections you looked at two very specific layouts. The HBox only organizes nodes horizontally across a Scene, whereas a VBox only organizes your nodes vertically down a Scene. The reason JavaFX can offer two otherwise limiting layouts and still be versatile is because these layouts can be nested within each other to create a more dynamic organization of nodes.

For example, let's say you want to create a Scene that has a TextBox followed by a Button horizontally. Then, directly under those nodes you want to have another TextBox followed by another Button. This is easily handled by nesting two HBox elements within a VBox.

The VBox organizes nodes vertically. In this case, you want to have two “groups” of nodes arranged vertically. You will have two HBox elements inside the VBox to represent the two horizontal groupings of nodes that are to be stacked on top of each other. Confused? Don't worry, this all will become much clearer to you after seeing the code.

To begin, create your VBox as follows:

```
VBox {  
    content: [  
        ]  
}
```

Within your VBox, create two separate HBox instances, like this:

```
VBox {  
    content: [HBox {  
        content: [  
            ]  
    }  
    HBox {  
        content: [  
            ]  
    }  
]  
}
```

These HBox instances give you a place to put the nodes you want to stack. Let's put one TextBox and one Button in each HBox:

```
VBox {  
    content: [HBox {  
        content: [TextBox {  
            text: "SampleText1"  
            columns: 12  
            selectOnFocus: true  
        } Button {  
            text: "Button1"  
            action: function () {  
            }  
        }  
    }]  
    HBox {  
        content: [TextBox {  
            text: "SampleText2"  
            columns: 12  
            selectOnFocus: true  
        } Button {  
            text: "Button2"  
            action: function () {  
            }  
        }  
    }  
}
```

That's really all you need to do to nest two different layouts within each other. This process is not limited to just two layouts. Multiple layouts can be nested within each

other to produce a more varied layout of nodes. The finished script file for this example produces a columned effect of two TextBox and two Button elements. Here is the full script:

```
/*
 * Chapter13.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/23/2010 - created
 *
 * Using JavaFX Layouts
 *
 */
package com.jfdimarzio.javafxforbeginners;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.layout.HBox;
import javafx.scene.control.TextBox;
import javafx.scene.control.Button;

/**
 * @author JFDiMarzio
 */
Stage {
    title: "UsingLayouts"
    onClose: function () {
    }
    scene: Scene {
        width: 200
        height: 200
        content: [VBox {
            content: [HBox {
                content: [TextBox {
                    text: "SampleText1"
                    columns: 12
                    selectOnFocus: true
                } Button {
                    text: "Button1"
                    action: function () {
                    }
                }]
            }]
        }]
    }
}
```

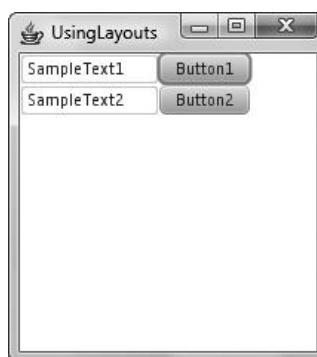
```
    HBox {
        content: [TextBox {
            text: "SampleText2"
            columns: 12
            selectOnFocus: true
        } Button {
            text: "Button2"
            action: function () {
            }
        }]
    }
}
```

Compile and run this script. You will see the layout shown in Figure 13-4.

Now you can experiment with the ClipView, Flow, Stack, and Tile layouts. Try to nest these layouts to produce custom ones that are original and engaging.

## Try This Using Other Layouts

Using the examples given in this chapter, create a JavaFX application that uses a Flow, Stack, or Tile layout to display three images. Notice how each layout will change the display of the images.



---

Figure 13-4 Nested layout

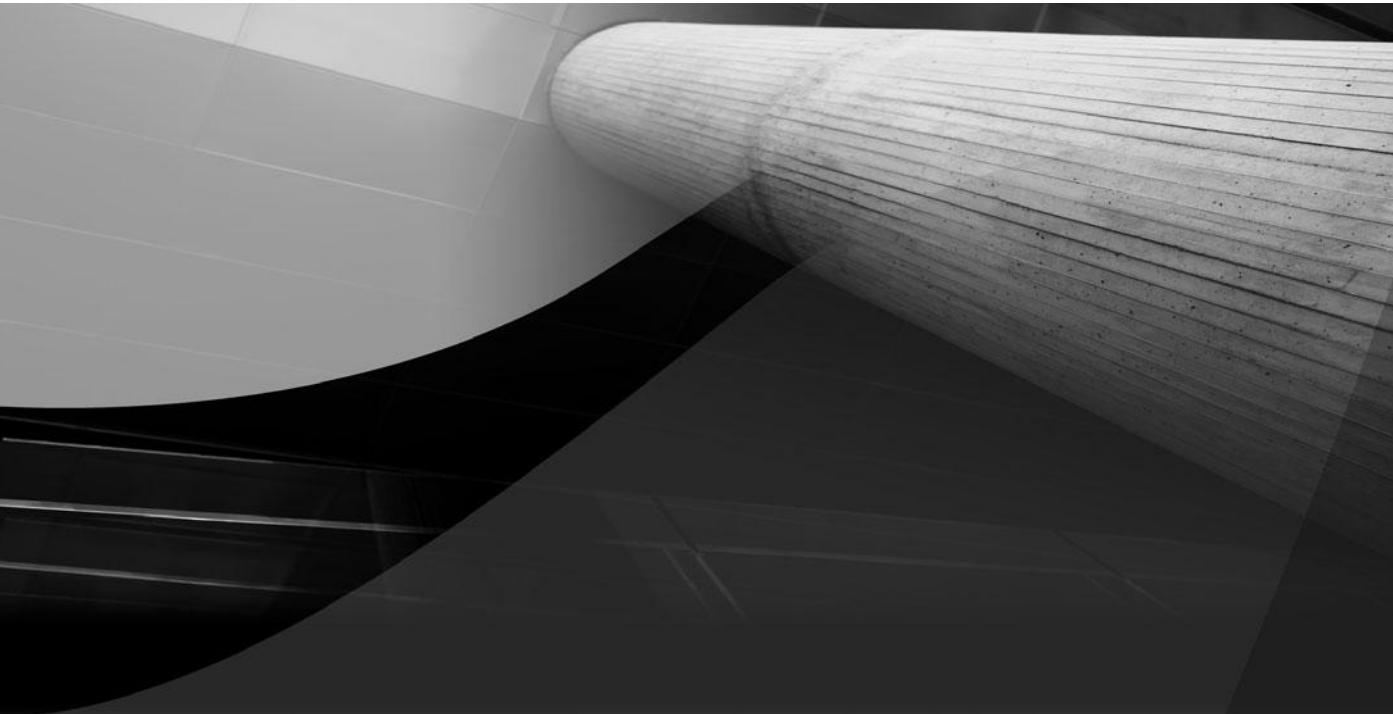
In the final chapter of this book, you learn about using CSS to add even more originality to your UI.



## Chapter 13 Self Test

1. What layout organizes your nodes horizontally across a Scene?
2. True or false? The HBox is located in the javafx.scene.HBox package.
3. What property holds the nodes for a layout to organize?
4. True or false? You must be sure to set the x and y coordinates of each node you place in a layout.
5. Can effects be applied to layouts?
6. What layout organizes nodes vertically down a Scene?
7. What is the name given to layouts that are combined to produce a new layout?
8. True or false? For layouts to be nested, one must inherit from the other.
9. True or false? Only two layouts can be nested.
10. Name four layouts other than the VBox and HBox.

*This page intentionally left blank*



# Chapter 14

## Style Your JavaFX with CSS

## Key Skills & Concepts

- Adding CSS files to your packages
  - Using CSS classes
  - Accessing Node properties
- 

In this chapter you learn how to use Cascading Style Sheets (CSS) to easily change the look and feel of your JavaFX applications. If you are not fully aware of what CSS is, this very quick refresher should help.

CSS is a styling language that allows you to separate the styling elements of an object from the object itself. Although CSS was created long before JavaFX, JavaFX includes the ability to use this styling language. In fact, the JavaFX CSS is based on the W3C version 2.1 CSS standard. That means you can create all your objects—or nodes in JavaFX—without any care for their placement, look, or feel. All you have to do is define the functionality of your nodes. Then, either in the style property of the nodes or in a separate file altogether, you can define styles that change the placement, look, and feel of your nodes.

An important feature of CSS you should note here is that all your CSS styles can be contained in a file separate from your script. Keeping your scripts and .css files separate allows you to change your styles—and even change the complete look and feel of your application—without needing to change or recompile your script. What does this mean to you? You can change the look and feel of an application you've designed by modifying the .css file, while leaving your application's script untouched.

## Ask the Expert

**Q:** Is there a difference between the CSS used in JavaFX and the CSS used on web pages?

**A:** Yes. JavaFX supports new JavaFX-specific elements. It is unclear how many, if any, of the standard CSS elements JavaFX will support in the future.

The remainder of this chapter teaches you how to leverage this very useful tool in your scripts. Before you begin, set up a new, empty JavaFX script named **Chapter14.fx**, as follows:

```
/*
 * Chapter14.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/25/2010 - created
 *
 * Styling JavaFX with CSS
 *
 */
package com.jfdimarzio.javafxforbeginners;

/**
 * @author JFDiMarzio
 */
```

In the first section of this chapter, you learn how to add a style sheet to your JavaFX package for use in your script.

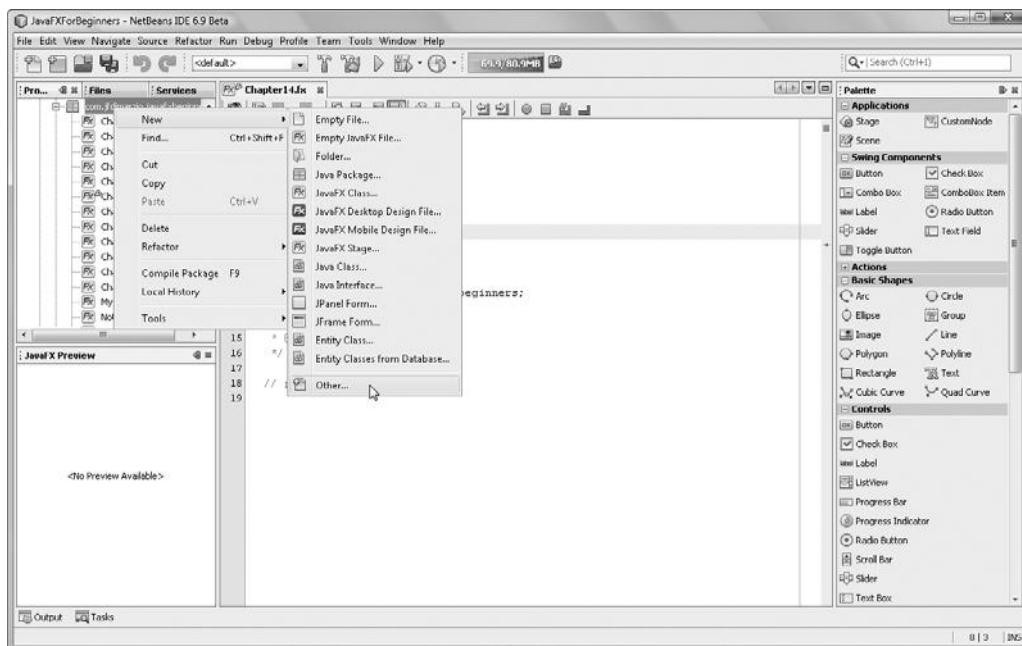
## Adding a Style Sheet to Your Packages

Although you can add styles directly to your script file, this is not recommended and in essence defeats the purpose for being able to separate your styles from your scripts to begin with. Therefore, you are going to learn how to create a separate .css file, add it to your package, and create some usable styles.

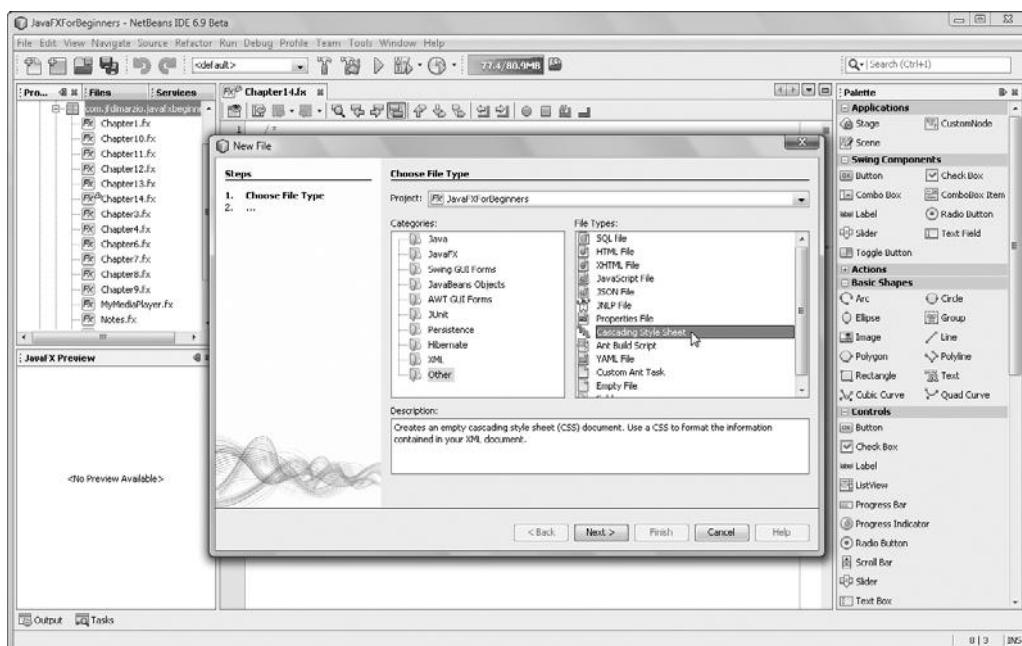
The first step is to right-click your package name from the Projects view of NetBeans (on the left side of the IDE). Select New | Other from the context menu, as shown in Figure 14-1.

This opens the New File Wizard. In the wizard, you want to select Other from the Categories area and then Cascading Style Sheet from the File Types area, as shown in Figure 14-2.

Finally, click the Next button, name your file **default**, and then click the Finish button, as shown in Figure 14-3.



**Figure 14-1** Select New | Other from the context menu



**Figure 14-2** Choosing Cascading Style Sheet

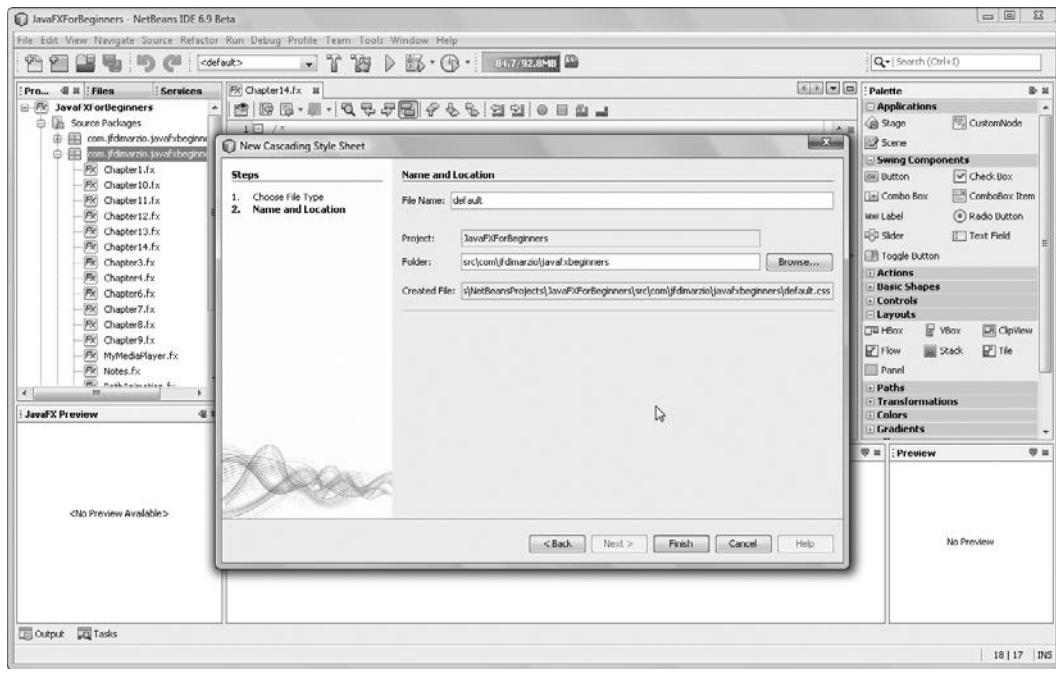


Figure 14-3 Finalizing your file

You now have a file named default.css in your list of package files. If it is not open already, open your style sheet. It should appear as follows:

```
/*
 Document      : default
 Created on   : Jul 31, 2010, 11:35:22 AM
 Author       : JFDiMarzio
 Description:
 Purpose of the stylesheet follows.

*/
/* 
 TODO customize this sample style
 Syntax recommendation http://www.w3.org/TR/REC-CSS2/
*/
root{
    display:block;
}
```

Notice how the CSS file contains a comments section, much like your JavaFX script files. This comments section is followed by the styles. When you create a new style sheet using the New File Wizard, one style is automatically created for you. This style contains one selector and one declaration. This is a simple style that states that anything in the root is visible.

**TIP**

Keep in mind, the purpose of this chapter is not to teach you CSS as a language. Rather you are just learning how to use CSS in relationship to JavaFX. If you need more instruction on CSS and how to use CSS properties, try using one of the many online resources available before you proceed with this chapter.

With the style sheet created and inserted into the working package, you can focus on creating your first style. In the next section you create a style that can be applied to the Label nodes in a future JavaFX script.

## Creating a Style

JavaFX will automatically recognize any CSS style class created with the name of a node. For example, if you wanted to create a CSS class that applies to all Label nodes, you would create a style class named .label. Let's create a .label class now. You will use this class in your script later.

```
.label{  
}
```

In this class, you change the Label node's font color to red and the font to Courier 14 pt. You need to add the correct properties to the style class for changing the font and font color. Luckily, JavaFX also recognize CSS style properties that are written to directly access Node properties.

The Node property for changing the color of a font in a label is `textFill`. To access this property from the CSS, you need to add “-fx-” as a prefix and separate each word with a hyphen (-). Thus, the style declaration would look like this:

```
.label{  
    -fx-text-fill: red;  
}
```

This style states that any Label node will have its `textFill` property set to RED. Let's create one more declaration for the Label node, and then you can apply this style sheet to a script.

To change the font of the Label node, use the `-fx-font` style, as follows:

```
.label{  
    -fx-text-fill: red;  
    -fx-font: bold 14pt "Courier";  
}
```

In total, this style changes the font of all labels to a red, bold, 14-point Courier. The full CSS file should look like this:

```
/*  
    Document      : default  
    Created on   : Jul 31, 2010, 11:35:22 AM  
    Author        : JFDiMarzio  
    Description:  
        Purpose of the stylesheet follows.  
*/  
  
/*  
    TODO customize this sample style  
    Syntax recommendation http://www.w3.org/TR/REC-CSS2/  
*/  
root{  
    display:block;  
}  
  
.label{  
    -fx-text-fill: red;  
    -fx-font: bold 14pt "Courier";  
}
```

In the next section you apply this full CSS file to a JavaFX script.

## Using Your Styles

In the previous sections you learned how to create a separate CSS file. You then created a style within that CSS file to be applied to all Label nodes. Let's now open Chapter14.fx and create a label that can use this new CSS file.

First, add a Stage and Scene to your file, as follows:

```
import javafx.stage.Stage;  
import javafx.scene.Scene;  
  
/**  
 * @author JFDiMarzio  
 */
```

```
Stage {  
    title: "Using CSS Styles"  
    onClose: function () {  
    }  
    scene: Scene {  
        width: 300  
        height: 200  
        content: []  
    }  
}
```

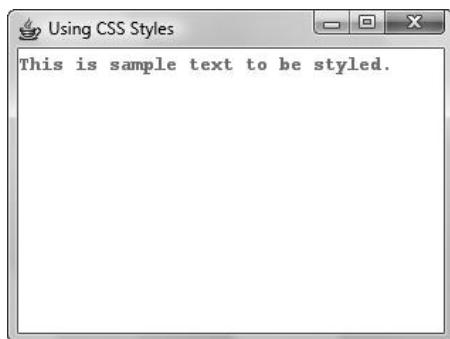
To use the style sheet in this script, you need to apply it to the Scene. The Scene has a property called `styleSheets`. Notice that the name of the property is plural. This is because you can apply multiple style sheets to a single Scene.

Use the `__DIR__` constant to indicate the location of the style sheet, as follows:

```
Stage {  
    title: "Using CSS Styles"  
    onClose: function () {  
    }  
  
    scene: Scene {  
        stylesheets: ["{__DIR__}default.css"]  
        width: 300  
        height: 200  
        content: []  
    }  
}
```

Now, simply add a Label node to the Scene's content:

```
import javafx.scene.control.Label;  
Stage {  
    title: "Using CSS Styles"  
    onClose: function () {  
    }  
  
    scene: Scene {  
        stylesheets: ["{__DIR__}default.css"]  
        width: 300  
        height: 200  
        content: [Label {  
            text: "This is sample text to be styled."  
        }  
    ]  
}
```



**Figure 14-4** Label with style applied

Notice that within the script you are not modifying the Label node in any way. Normally this would result in a label with standard black font. However, because you applied the default.css file, the font will end up being red, bold, and Courier.

Compile and run this script. You will see a result like that shown in Figure 14-4.

In the final section of this chapter, you learn about creating style classes that can be applied independent of the node.

## Creating Independent Style Classes

Open the default.css file once more. You are going to add a style class to this file that can be applied to any node, regardless of its type. That is, whereas you created a style previously that would only be applied to all labels, you will now create a style class that can be applied to any node you want.

Let's create a class that rotates any node by 90 degrees. Add the following class to your default.css file:

```
.rotate{  
    -fx-rotate:90;  
}
```

Notice here that the Node property for rotating a node is called rotate. Therefore, the style sheet declaration for this is -fx-rotate. Your full default.css should now look like this:

```
/*  
Document : default  
Created on : Jul 31, 2010, 11:35:22 AM  
Author   : JFDiMarzio
```

```
Description:  
    Purpose of the stylesheet follows.  
*/  
  
/*  
    TODO customize this sample style  
    Syntax recommendation http://www.w3.org/TR/REC-CSS2/  
*/  
root{  
    display:block;  
}  
  
.label{  
    -fx-text-fill: red;  
    -fx-font: bold 14pt "Courier";  
}  
.rotate{  
    -fx-rotate:90;  
}
```

Now you need to modify your Chapter14.fx script to use this new class. If it is not already open, open the Chapter14.fx script.

The .rotate class you just created is an independent class that you need to call specifically if you want to use it—unlike the .label class, which will inherently be applied to all labels. The styleClass property of a node allows you to specify a class within the current style sheet that you want to apply to the node. Modify the Label code in your Chapter14.fx script to include the styleClass property:

```
Stage {  
    title: "Using CSS Styles"  
    onClose: function () {  
    }  
  
    scene: Scene {  
        stylesheets: ["{__DIR__}default.css"]  
        width: 300  
        height: 200  
        content: [Label {  
            styleClass:  
                text: "This is sample text to be styled."  
        }  
    ]  
}
```

Assign the name of the class (rotate) to the styleClass property, as shown in the full script:

```
/*
 * Chapter14.fx
 *
 * v1.0 - J. F. DiMarzio
 *
 * 6/25/2010 - created
 *
 * Styling JavaFX with CSS
 *
 */
package com.jfdimarzio.javafxforbeginners;

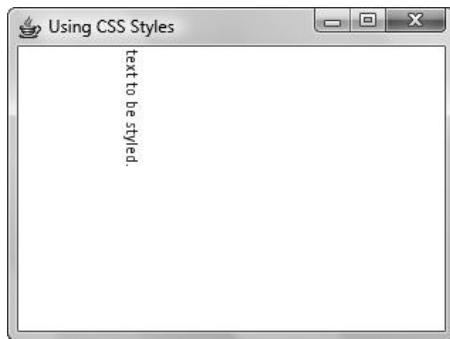
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;

/**
 * @author JFDiMarzio
 */
Stage {
    title: "Using CSS Styles"
    onClose: function () {
    }

    scene: Scene {
        stylesheets: ["{__DIR__}default.css"]
        width: 300
        height: 200
        content: [Label {
            styleClass: "rotate"
            text: "This is sample text to be styled."
        }]
    }
}
```

Compile and run this script, with the new default.css. You will see a change in your label's text, like that shown in Figure 14-5.

Take a good look at the outcome of running this script. There is a very important feature to note about using style sheets. Notice that the text, although it is now rotated by



---

**Figure 14-5** Label with the `.rotate` class

90 degrees, is no longer red or 14-point bold Courier. This is because any style classes specifically assigned to a node using the `styleClass` property will override any style applied to the node as a whole. Therefore, the style declarations in the `.label` class are being overridden by those in the `.rotate` class. This means that only the style declarations from `.rotate` are being applied to the label. None of the style declarations from `.label` are being applied.

## Try This Experimenting with Styles

Experiment with styles and nodes. Using the examples covered in this chapter, create a style class that changes the position or color of a node.

---

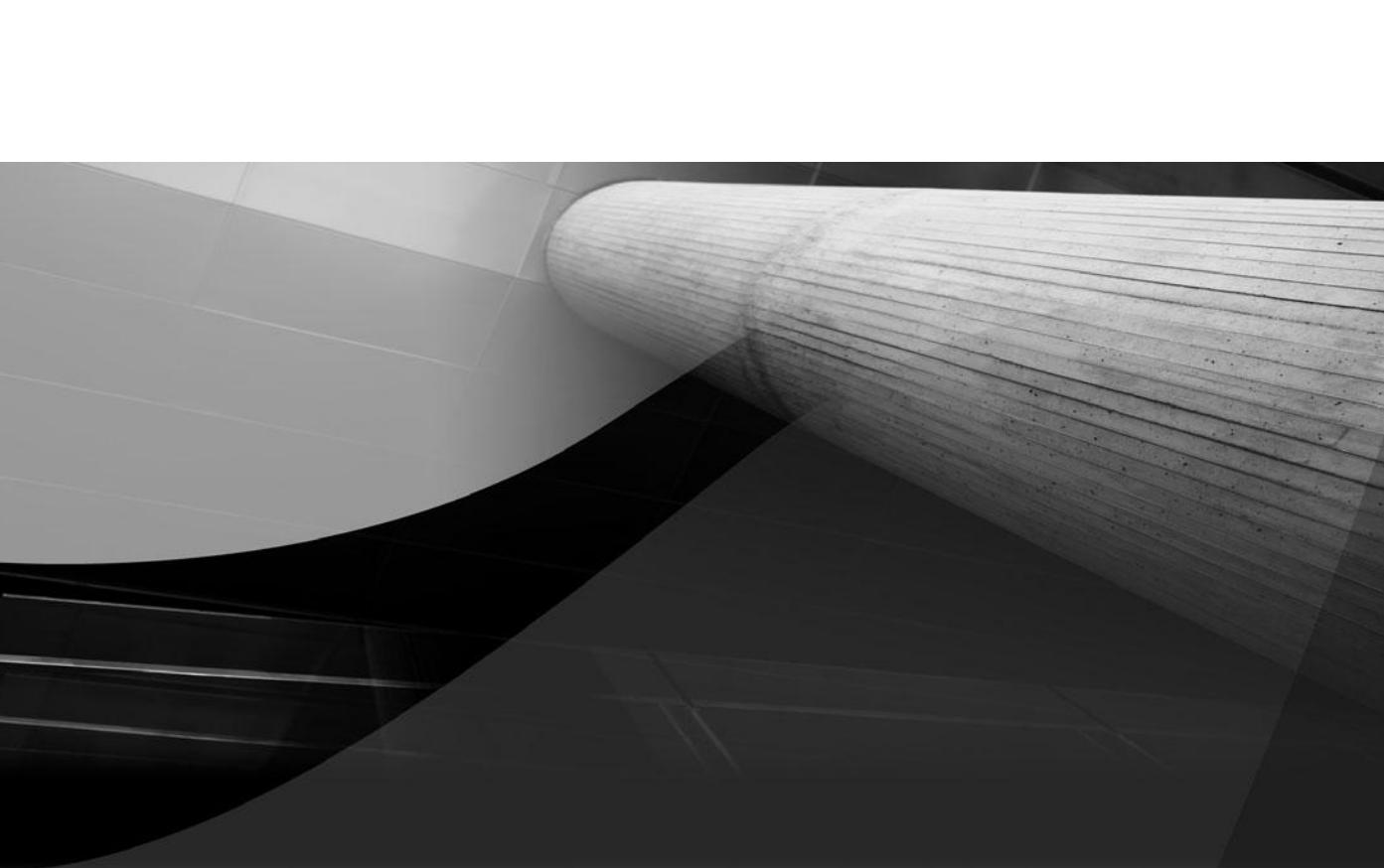


## Chapter 14 Self Test

1. What is Cascading Style Sheets (CSS)?
2. What file extension is used for Cascading Style Sheets?
3. What wizard helps you create and add a CSS to your package?
4. If you use the wizard to create your CSS, what class is added by default?

5. True or false? To create a CSS class that applies to all nodes of a certain type, the name of the class should be the name of the node type in lowercase.
6. What prefix is added to every Node property to call it from a CSS class?
7. What property of Scene will let you apply a style sheet to your script?
8. True or false? You can only add one style sheet to a Scene.
9. What Node property allows you to assign a specific CSS class to a node?
10. True or false? If you have a node with a node-applied CSS class and a styleClass property, the style in the styleClass will override that in the node-applied style.

*This page intentionally left blank*



# Appendix A

## Deploying JavaFX

Much of this book was spent covering the basics of JavaFX development. You have learned how to navigate JavaFX scripting and create some very compelling applications.

Admittedly, you have only scratched the surface of what is available to you in JavaFX, which is a very deep language with a lot to offer. No doubt, you have some questions about some of the things you have learned. Therefore, this book's appendixes attempt to fill in some of the holes that may have been left in the lessons of this book.

## Deploying JavaFX

Throughout this book you have created multiple JavaFX scripts that you can execute from within NetBeans. You have written some very useful scripts in your short time with the language, but one issue remains: You need to be able to deploy the scripts you write so that others can execute your applications.

The following steps teach you a quick-and-easy way to deploy your JavaFX applications using NetBeans.

### **NOTE**

In this example you will be using Chapter12.fx. If you do not have Chapter12.fx from the example in Chapter 12, any of the book's examples will work. In the event you have not retained any of the examples in the book, you may want to complete one before following these steps.

The first step in deploying JavaFX from NetBeans is to right-click your project in NetBeans and go to the project's Properties dialog box. From there, select the Run properties, as shown in Figure A-1.

In the Run properties, you just want to confirm that the Main class is the .fx script file you want to execute. The .fx file listed in the Main Class section is the file that will be compiled and deployed during this process. Once you have confirmed or set the correct .fx script file as the Main class, you can close the project's Properties dialog box.

The next step in deploying your JavaFX application is to build your project. The build process compiles the scripts so that they are executable outside the NetBeans IDE. To build your project, right-click your project once again and select Build Project, as shown in Figure A-2.

When the build process completes, you will have a fully executable set of files. These files are placed in the dist folder of your NetBeans project. The contents of the dist folder are shown in Figure A-3.

Notice that one of the files created for you during the build process is an HTML file with the name of your project. This file is a sample distribution file that contains all the

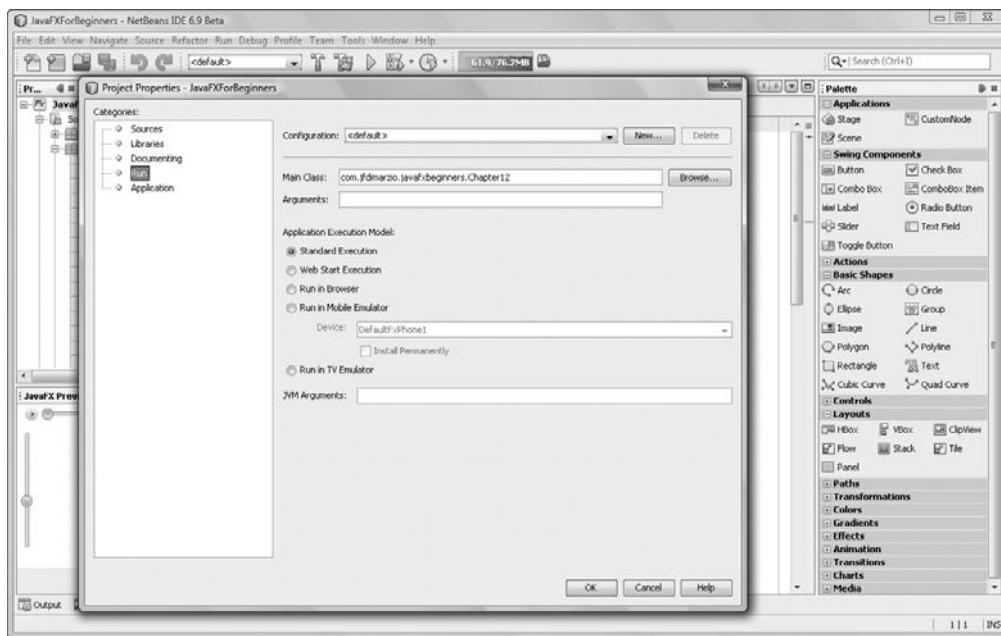


Figure A-1 Run properties

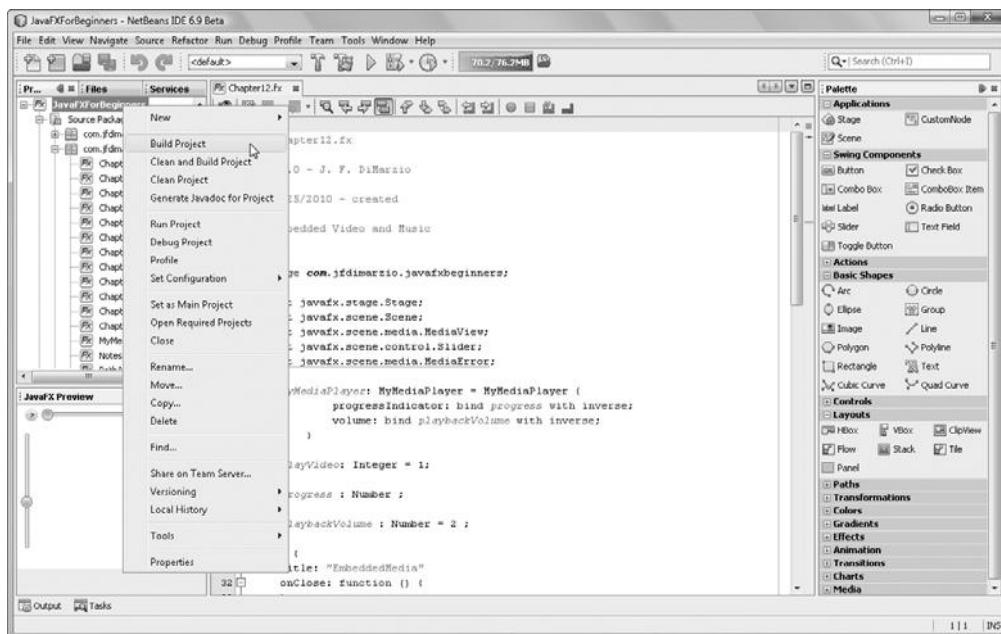
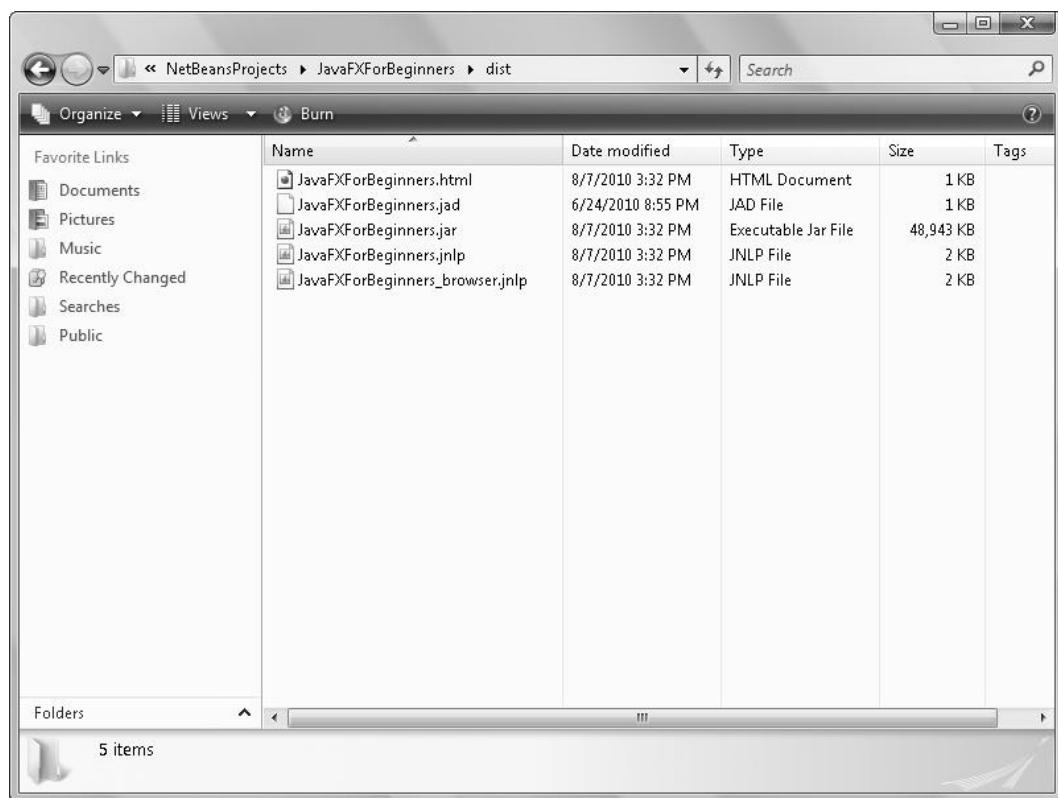


Figure A-2 Build Project



**Figure A-3** The dist folder

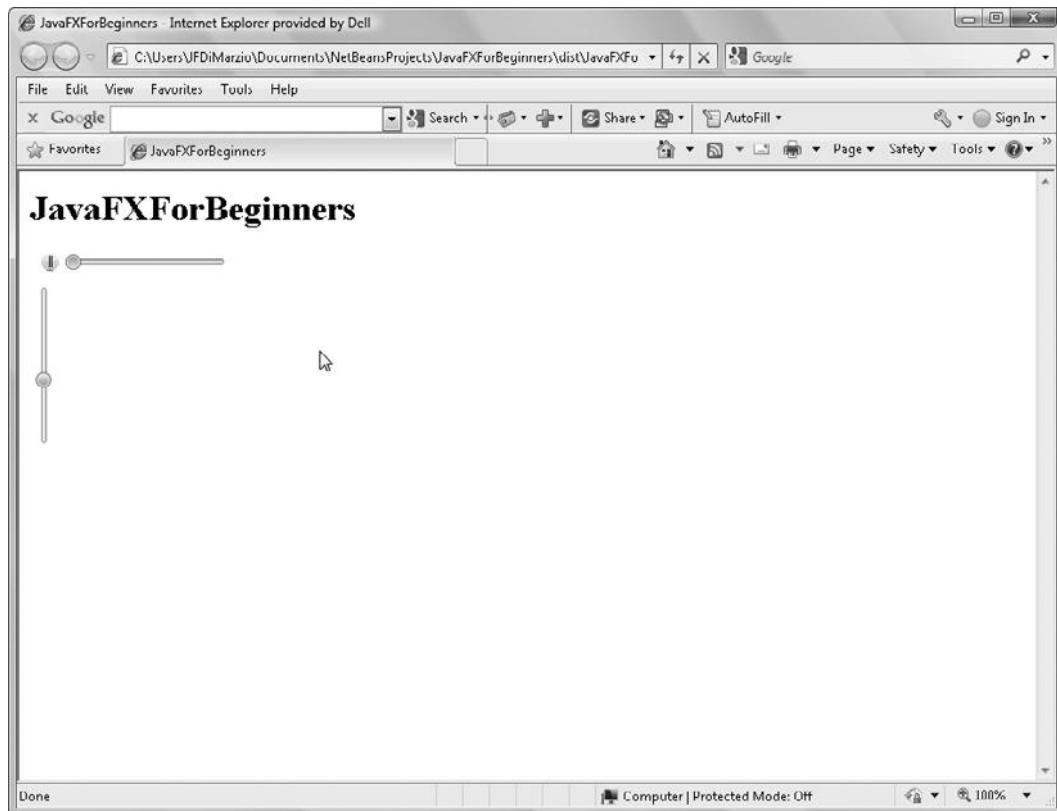
information needed to distribute your app successfully. If you open the file, you will see that it contains the following code:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>JavaFXForBeginners</title>
</head>
<body>
<h1>JavaFXBeginners</h1>
<script src="http://dl.javafx.com/1.3/dtfx.js"></script>
<script>
    javafx(
    {
        archive: "JavaFXBeginners.jar",
        draggable: true,
        width: 200,
```

```
height: 200,  
code: "com.jfdimarzio.javafxbeginners.Chapter12",  
extPackages: "javafx.ext.swing",  
name: "JavaFXBeginners"  
}  
};  
</script>  
</body>  
</html>
```

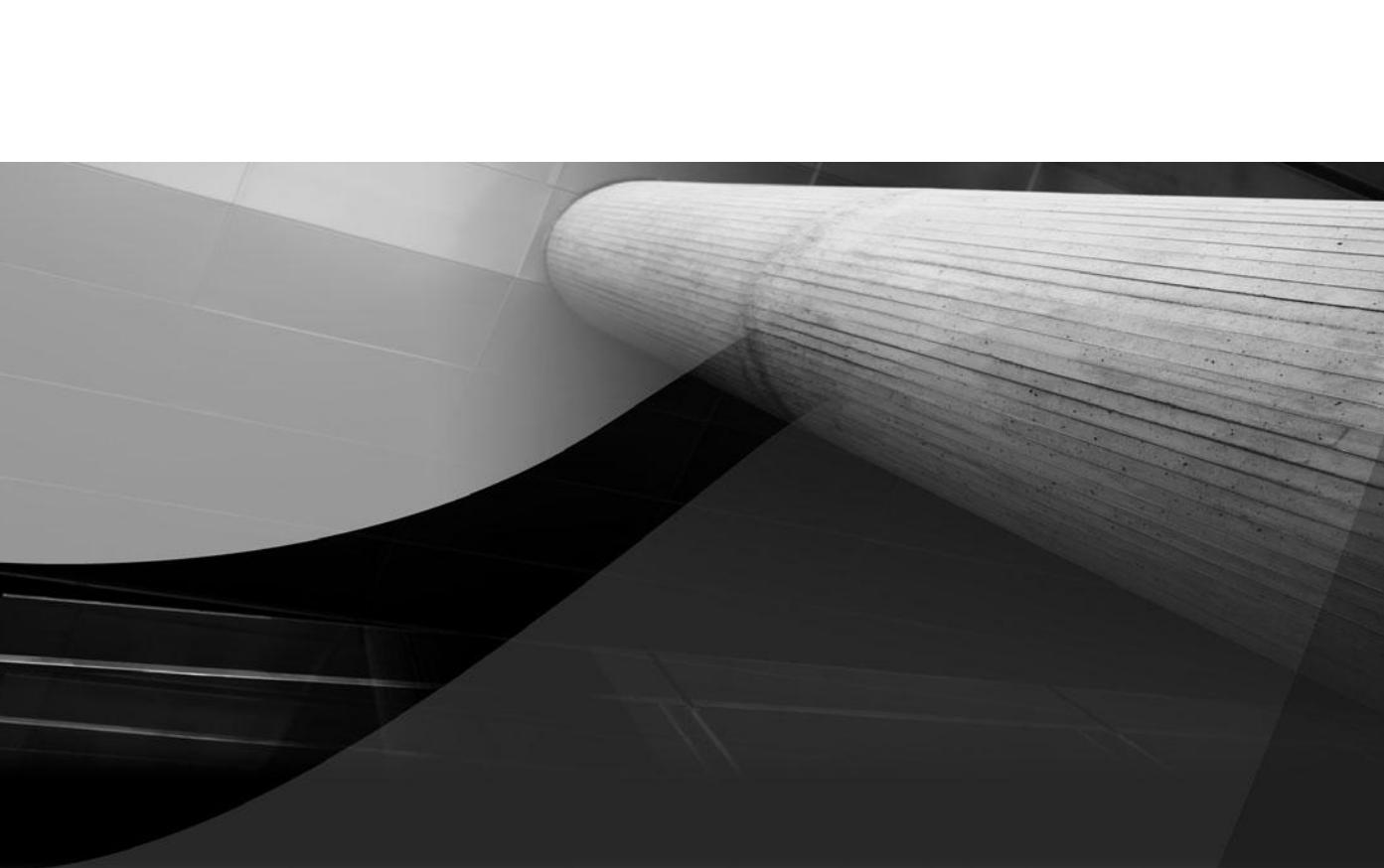
Within the script section of the HTML is a javafx script that allows the compiled project to run when this page is opened. Open the HTML file with your browser and you should see the fully functional application, as shown in Figure A-4.

In this short appendix you learned how to compile and build your script files so that they can be executed outside of the NetBeans IDE. As you can see, the distribution process is very easy—which only adds to the usefulness and accessibility of JavaFX.



**Figure A-4** A distributed app

*This page intentionally left blank*



# Appendix B

## Node Property Reference

The purpose of this appendix is to supply some information on certain topics that may not have been fully provided in the chapters of this book. The first section of this appendix provides a reference of the uncovered Node properties.

## Node Properties

Much of what you have learned in this book was based on the JavaFX Node class. From using the mouse events, to creating custom nodes, JavaFX is really a language built around its Node. Most of the objects you have worked with are built around the Node class and therefore inherit all of Node's properties.

In working with the objects that inherit from Node, you have really only worked with a handful of Node's properties. In an effort to get better acquainted with the language, you should familiarize yourself with all of Node's properties—especially because they are present in a majority of the objects you will work with.

Luckily, JavaFX docs provide a great reference for the properties of Node. Table B-1 lists all the properties inherited through Node.

Property	Type	Default	Description
blocksMouse	Boolean	false	If true, this property consumes mouse events in this Node and does not send them to other nodes further up the scene graph.
boundsInLocal	Rectangle2D		The rectangular bounds of this Node in the Node's untransformed local coordinate space.
boundsInParent	Rectangle2D		The rectangular bounds of this Node, which include its transforms.
boundsInScene	Rectangle2D		The rectangular bounds of this Node in the coordinate space of the javafx.scene.Scene containing the Node.
cache	Boolean	false	A performance hint to the system to indicate that this Node should be cached as a bitmap.
clip	Node	null	Specifies a Node to use to define the clipping shape for this Node.
cursor	Cursor	null	Defines the mouse cursor for this Node and subnodes.
disable	Boolean	false	Sets the individual disabled state of this Node.
disabled	Boolean	false	Indicates whether or not this Node is disabled.

**Table B-1** Node Properties

Property	Type	Default	Description
effect	Effect	null	Specifies an effect to apply to this Node.
focusable	Boolean	true	Specifies whether this Node should accept input focus.
focused	Boolean	false	Indicates whether this Node is the current input focus owner.
hover	Boolean	false	Indicates whether or not this Node is being hovered over.
id	String	empty string	The ID of this Node.
layoutBounds	Rectangle2D		The rectangular bounds that should be used in calculations for both manual and automatic layout of this Node.
onKeyPressed	function(:KeyEvent):Void		Defines a function to be called when this Node has input focus and a key has been pressed.
onKeyReleased	function(:KeyEvent):Void		Defines a function to be called when this Node has input focus and a key has been released.
onKeyTyped	function(:KeyEvent):Void		Defines a function to be called when this Node has input focus and a key has been typed.
onMouseClicked	function(:MouseEvent):Void		Defines a function to be called when a mouse button has been clicked (pressed and released) on this Node.
onMouseDragged	function(:MouseEvent):Void		Defines a function to be called when a mouse button is pressed on this Node and then dragged.
onMouseEntered	function(:MouseEvent):Void		Defines a function to be called when the mouse enters this Node.
onMouseExited	function(:MouseEvent):Void		Defines a function to be called when the mouse exits this Node.
onMouseMoved	function(:MouseEvent):Void		Defines a function to be called when mouse cursor moves within this Node but no buttons have been pressed.
onMousePressed	function(:MouseEvent):Void		Defines a function to be called when a mouse button has been pressed on this Node.
onMouseReleased	function(:MouseEvent):Void		Defines a function to be called when a mouse button has been released on this Node.
onMouseWheelMoved	function(:MouseEvent):Void		Defines a function to be called when the mouse scroll wheel has moved.

**Table B-1** Node Properties (*continued*)

Property	Type	Default	Description
opacity	Number	1.0	Specifies how opaque (that is, solid) the Node appears.
parent	Node	null	The parent of this Node.
pressed	Boolean	false	Indicates whether or not the Node is pressed.
rotate	Number	0.0	Defines the angle of rotation about the Node's center, measured in degrees.
scaleX	Number	1.0	Defines the factor by which coordinates are scaled about the center of the object along the x axis of this Node.
scaleY	Number	1.0	Defines the factor by which coordinates are scaled about the center of the object along the y axis of this Node.
scene	Scene	null	The Scene that this Node is part of.
style	String	empty string	A string representation of the CSS style associated with this specific Node.
styleClass	String	empty string	A String identifier that can be used to logically group Nodes, specifically for an external style engine.
transforms	Transform[]	empty	Defines the sequence of javafx.scene.transform.Transform objects to be applied to this Node.
translateX	Number	0	Defines the x coordinate of the translation that is added to the transformed coordinates of this Node.
translateY	Number	0	Defines the y coordinate of the translation that is added to the transformed coordinates of this Node.
visible	Boolean	true	Specifies whether this Node and any subnodes should be rendered as part of the scene graph.

**Table B-1** Node Properties (*continued*)

The next section covers the mouse events not covered in Chapter 9.

## Mouse Events

In Chapter 9, you learned about some of the mouse events provided in JavaFX. However, in an effort to cover the more logical events for the basic user, some events were skipped. Table B-2 contains a list of all the mouse events.

Event	Type	Description
altDown	Boolean	Whether or not the Alt modifier is down on this event.
button	MouseButton	Which, if any, of the mouse buttons has changed state.
clickCount	Integer	The number of mouse clicks associated with this event.
controlDown	Boolean	Whether or not the Control modifier is down on this event.
dragAnchorX	Number	The x position of the event that initiated the most recent press event if the MouseEvent is part of a press-drag-release gesture. Otherwise, the value is 0.
dragAnchorY	Number	The y position of the event that initiated the most recent press event only if the MouseEvent is part of a press-drag-release gesture. Otherwise, the value is 0.
dragX	Number	The x offset of the event relative to the most recent press event if the MouseEvent is part of a press-drag-release gesture. Otherwise, the value is 0.
dragY	Number	The y offset of the event relative to the most recent press event only if the MouseEvent is part of a press-drag-release gesture. Otherwise, the value is 0.
metaDown	Boolean	Whether or not the Meta modifier is down on this event.
middleButtonDown	Boolean	Set to true if the middle button (button 2) is currently pressed.
node	Node	The Node on which this event has occurred.
popupTrigger	Boolean	Whether or not this mouse event is the pop-up menu trigger event for the platform.
primaryButtonDown	Boolean	Set to true if the primary button (button 1, usually the left button) is currently pressed.
sceneX	Number	Horizontal x position of the event relative to the origin of the Scene that contains the MouseEvent's node.
sceneY	Number	Vertical y position of the event relative to the origin of the Scene that contains the MouseEvent's node.
screenX	Number	Absolute horizontal x position of the event.
screenY	Number	Absolute vertical y position of the event.
secondaryButtonDown	Boolean	Set to true if the secondary button (button 3, usually the right button) is currently pressed.
shiftDown	Boolean	Whether or not the Shift modifier is down on this event.
wheelRotation	Number	Number of clicks the mouse wheel was rotated.
x	Number	Horizontal x position of the event relative to the origin of the MouseEvent's node.
y	Number	Vertical y position of the event relative to the origin of the MouseEvent's node.

**Table B-2** Mouse Events

The next section covers another key piece of data that was only lightly covered in Chapter 9: the key codes.

## Key Codes

The key codes are what you use to identify which key has been pressed during a KeyEvent. Unfortunately, there was not enough space to logically cover all 200+ key codes in Chapter 9. However, a reference of these key codes can prove to be very valuable. Therefore, Table B-3 lists all the key codes for JavaFX.

### **NOTE**

Not all these keys pertain to a computer's QWERTY keyboard. There are keys for devices such as media players, cell phones, and others.

Value	Type	Description
VK_0	public static final	0 number key
VK_1	public static final	1 number key
VK_2	public static final	2 number key
VK_3	public static final	3 number key
VK_4	public static final	4 number key
VK_5	public static final	5 number key
VK_6	public static final	6 number key
VK_7	public static final	7 number key
VK_8	public static final	8 number key
VK_9	public static final	9 number key
VK_A	public static final	A
VK_B	public static final	B
VK_C	public static final	C
VK_D	public static final	D
VK_E	public static final	E
VK_F	public static final	F
VK_G	public static final	G
VK_H	public static final	H
VK_I	public static final	I
VK_J	public static final	J
VK_K	public static final	K

**Table B-3** Key Codes

<b>Value</b>	<b>Type</b>	<b>Description</b>
VK_L	public static final	L
VK_M	public static final	M
VK_N	public static final	N
VK_O	public static final	O
VK_P	public static final	P
VK_Q	public static final	Q
VK_R	public static final	R
VK_S	public static final	S
VK_T	public static final	T
VK_U	public static final	U
VK_V	public static final	V
VK_W	public static final	W
VK_X	public static final	X
VK_Y	public static final	Y
VK_Z	public static final	Z
VK_ACCEPT	public static final	Constant for the Accept or Commit function key
VK_ADD	public static final	Plus sign
VK AGAIN	public static final	Repeat key used on some devices
VK_ALL_CANDIDATES	public static final	Constant for the All Candidates function key
VK_ALPHANUMERIC	public static final	Constant for the Alphanumeric function key
VK_ALT	public static final	ALT key
VK_ALT_GRAPH	public static final	Constant for the AltGraph function key
VK_AMPERSAND	public static final	Ampersand
VK_ASTERISK	public static final	Asterisk
VK_AT	public static final	Constant for the @ key
VK_BACK	public static final	Back
VK_BACK_QUOTE	public static final	Opening quote
VK_BACK_SLASH	public static final	Constant for the backslash key (\)
VK_BACK_SPACE	public static final	BACKSPACE
VK_BEGIN	public static final	Constant for the Begin key
VK_BRACELEFT	public static final	Left brace
VK_BRACERIGHT	public static final	Right brace
VK_CANCEL	public static final	Cancel

**Table B-3** Key Codes (continued)

<b>Value</b>	<b>Type</b>	<b>Description</b>
VK_CAPS	public static final	CAPS LOCK
VK_CHANNEL_DOWN	public static final	Channel Down
VK_CHANNEL_UP	public static final	Channel Up
VK_CIRCUMFLEX	public static final	Constant for the caret (^) key
VK_CLEAR	public static final	Clear
VK_CLOSE_BRACKET	public static final	Constant for the closing bracket key ()]
VK_CODE_INPUT	public static final	Constant for the Code Input function key
VK_COLON	public static final	Constant for the colon key (:)
VK_COLORED_KEY_0	public static final	Colored 0 key
VK_COLORED_KEY_1	public static final	Colored 1 key
VK_COLORED_KEY_2	public static final	Colored 2 key
VK_COLORED_KEY_3	public static final	Colored 3 key
VK_COMMA	public static final	Constant for the comma key (,)
VK_COMPOSE	public static final	Constant for the input method on/off key
VK_CONTEXT_MENU	public static final	Constant for the Microsoft Windows context menu key
VK_CONTROL	public static final	CTRL key
VK_CONVERT	public static final	Constant for the Convert function key
VK_COPY	public static final	Copy
VK_CUT	public static final	Cut
VK_DEAD_ABOVEDOT	public static final	
VK_DEAD_ABOVERING	public static final	
VK_DEAD_ACUTE	public static final	
VK_DEAD_BREVE	public static final	
VK_DEAD_CARON	public static final	
VK_DEAD_CEDILLA	public static final	
VK_DEAD_CIRCUMFLEX	public static final	
VK_DEAD_DIAERESIS	public static final	
VK_DEAD_DOUBLEACUTE	public static final	
VK_DEAD_GRAVE	public static final	
VK_DEAD_IOTA	public static final	
VK_DEAD_MACRON	public static final	
VK_DEAD_OGONEK	public static final	

**Table B-3** Key Codes

<b>Value</b>	<b>Type</b>	<b>Description</b>
VK_DEAD_SEMIVOICED_SOUND	public static final	
VK_DEAD_TILDE	public static final	
VK_DEAD_VOICED_SOUND	public static final	
VK_DECIMAL	public static final	Decimal
VK_DELETE	public static final	DELETE
VK_DIVIDE	public static final	Division sign
VK_DOLLAR	public static final	Constant for the dollar sign key (\$)
VK_DOWN	public static final	Constant for the non-numeric keypad down-arrow key
VK_EJECT_TOGGLE	public static final	Eject button
VK_END	public static final	END
VK_ENTER	public static final	ENTER
VK_EQUALS	public static final	Constant for the equals sign key (=)
VK_ESCAPE	public static final	ESC key
VK_EURO_SIGN	public static final	Constant for the Euro currency sign key
VK_EXCLAMATION_MARK	public static final	Constant for the exclamation mark key (!)
VK_F1	public static final	Constant for the F1 function key
VK_F10	public static final	Constant for the F10 function key
VK_F11	public static final	Constant for the F11 function key
VK_F12	public static final	Constant for the F12 function key
VK_F13	public static final	Constant for the F13 function key
VK_F14	public static final	Constant for the F14 function key
VK_F15	public static final	Constant for the F15 function key
VK_F16	public static final	Constant for the F16 function key
VK_F17	public static final	Constant for the F17 function key
VK_F18	public static final	Constant for the F18 function key
VK_F19	public static final	Constant for the F19 function key
VK_F2	public static final	Constant for the F2 function key
VK_F20	public static final	Constant for the F20 function key
VK_F21	public static final	Constant for the F21 function key
VK_F22	public static final	Constant for the F22 function key
VK_F23	public static final	Constant for the F23 function key

**Table B-3** Key Codes (continued)

Value	Type	Description
VK_F24	public static final	Constant for the F24 function key
VK_F3	public static final	Constant for the F3 function key
VK_F4	public static final	Constant for the F4 function key
VK_F5	public static final	Constant for the F5 function key
VK_F6	public static final	Constant for the F6 function key
VK_F7	public static final	Constant for the F7 function key
VK_F8	public static final	Constant for the F8 function key
VK_F9	public static final	Constant for the F9 function key
VK_FAST_FWD	public static final	Fast Forward
VK_FINAL	public static final	Constant for input method support on Asian keyboards
VK_FIND	public static final	Find
VK_FULL_WIDTH	public static final	Constant for the Full-Width Characters function key
VK_GAME_A	public static final	Game Controller A
VK_GAME_B	public static final	Game Controller B
VK_GAME_C	public static final	Game Controller C
VK_GAME_D	public static final	Game Controller D
VK_GREATER	public static final	Greater than sign
VK_HALF_WIDTH	public static final	Constant for the Half-Width Characters function key
VK_HELP	public static final	Help
VK_HIRAGANA	public static final	Constant for the Hiragana function key
VK_HOME	public static final	HOME
VK_INFO	public static final	Info
VK_INPUT_METHOD_ON_OFF	public static final	Constant for the input method on/off key
VK_INSERT	public static final	Inset
VK_INVERTED_EXCLAMATION_MARK	public static final	Constant for the inverted exclamation mark key
VK_JAPANESE_HIRAGANA	public static final	Constant for the Japanese-Hiragana function key
VK_JAPANESE_KATAKANA	public static final	Constant for the Japanese-Katakana function key
VK_JAPANESE_ROMAN	public static final	Constant for the Japanese-Roman function key
VK_KANA	public static final	

**Table B-3** Key Codes

Value	Type	Description
VK_KANA_LOCK	public static final	Constant for the locking Kana function key
VK_KANJI	public static final	
VK_KATAKANA	public static final	Constant for the Katakana function key
VK_KP_DOWN	public static final	Constant for the numeric keypad down-arrow key
VK_KP_LEFT	public static final	Constant for the numeric keypad left-arrow key
VK_KP_RIGHT	public static final	Constant for the numeric keypad right-arrow key
VK_KP_UP	public static final	Constant for the numeric keypad up-arrow key
VK_LEFT	public static final	Constant for the non-numeric keypad left-arrow key
VK_LEFT_PARENTHESIS	public static final	Constant for the ( key
VK_LESS	public static final	Less than sign
VK_META	public static final	
VK_MINUS	public static final	Constant for the minus key (-)
VK_MODECHANGE	public static final	
VK_MULTIPLY	public static final	Multiplication sign
VK_MUTE	public static final	Mute
VK_NONCONVERT	public static final	Constant for the Don't Convert function key
VK_NUM_LOCK	public static final	NUM LOCK
VK_NUMBER_SIGN	public static final	Constant for the # key
VK_NUMPAD0	public static final	Number pad 0 key
VK_NUMPAD1	public static final	Number pad 1 key
VK_NUMPAD2	public static final	Number pad 2 key
VK_NUMPAD3	public static final	Number pad 3 key
VK_NUMPAD4	public static final	Number pad 4 key
VK_NUMPAD5	public static final	Number pad 5 key
VK_NUMPAD6	public static final	Number pad 6 key
VK_NUMPAD7	public static final	Number pad 7 key
VK_NUMPAD8	public static final	Number pad 8 key
VK_NUMPAD9	public static final	Number pad 9 key
VK_OPEN_BRACKET	public static final	Constant for the opening bracket key ([)
VK_PAGE_DOWN	public static final	PAGE DOWN
VK_PAGE_UP	public static final	PAGE UP
VK_PASTE	public static final	Paste

**Table B-3** Key Codes (continued)

<b>Value</b>	<b>Type</b>	<b>Description</b>
VK_PAUSE	public static final	PAUSE
VK_PERIOD	public static final	Constant for the period key (.)
VK_PLAY	public static final	Play
VK_PLUS	public static final	Constant for the + key
VK_POUND	public static final	Pound sign
VK_POWER	public static final	Power button
VK_PREVIOUS_CANDIDATE	public static final	Constant for the Previous Candidate function key
VK_PRINTSCREEN	public static final	PRINT SCREEN
VK_PROPS	public static final	
VK_QUOTE	public static final	Single quote
VK_QUOTEDBL	public static final	Double quote
VK_RECORD	public static final	Record
VK_REWIND	public static final	Rewind
VK_RIGHT	public static final	Constant for the non-numeric keypad right-arrow key
VK_RIGHT_PARENTHESIS	public static final	Constant for the ) key
VK_ROMAN_CHARACTERS	public static final	Constant for the Roman Characters function key
VK_SCROLL_LOCK	public static final	SCROLL LOCK
VK_SEMICOLON	public static final	Constant for the semicolon key (;)
VK_SEPARATOR	public static final	Constant for the Numpad Separator key
VK_SHIFT	public static final	SHIFT
VK_SLASH	public static final	Constant for the forward slash key (/)
VK_SOFTKEY_0	public static final	
VK_SOFTKEY_1	public static final	
VK_SOFTKEY_2	public static final	
VK_SOFTKEY_3	public static final	
VK_SOFTKEY_4	public static final	
VK_SOFTKEY_5	public static final	
VK_SOFTKEY_6	public static final	
VK_SOFTKEY_7	public static final	
VK_SOFTKEY_8	public static final	
VK_SOFTKEY_9	public static final	
VK_SPACE	public static final	Space

**Table B-3** Key Codes

Value	Type	Description
VK_STAR	public static final	Star key on the dialer pad
VK_STOP	public static final	Stop
VK_SUBTRACT	public static final	Minus sign
VK_TAB	public static final	TAB
VK_TRACK_NEXT	public static final	Track Skip key
VK_TRACK_PREV	public static final	Previous Track key
VK_UNDEFINED	public static final	Used to indicate that the key code is unknown
VK_UNDERSCORE	public static final	Constant for the underscore key (_)
VK_UNDO	public static final	Undo
VK_UP	public static final	Constant for the non-numeric keypad up-arrow key
VK_VOLUME_DOWN	public static final	Volume Down
VK_VOLUME_UP	public static final	Volume Up
VK_WINDOWS	public static final	Constant for the Microsoft Windows "Windows" key

**Table B-3** Key Codes (continued)

The next section covers the Media Player properties not covered in Chapter 12.

## MediaPlayer Properties

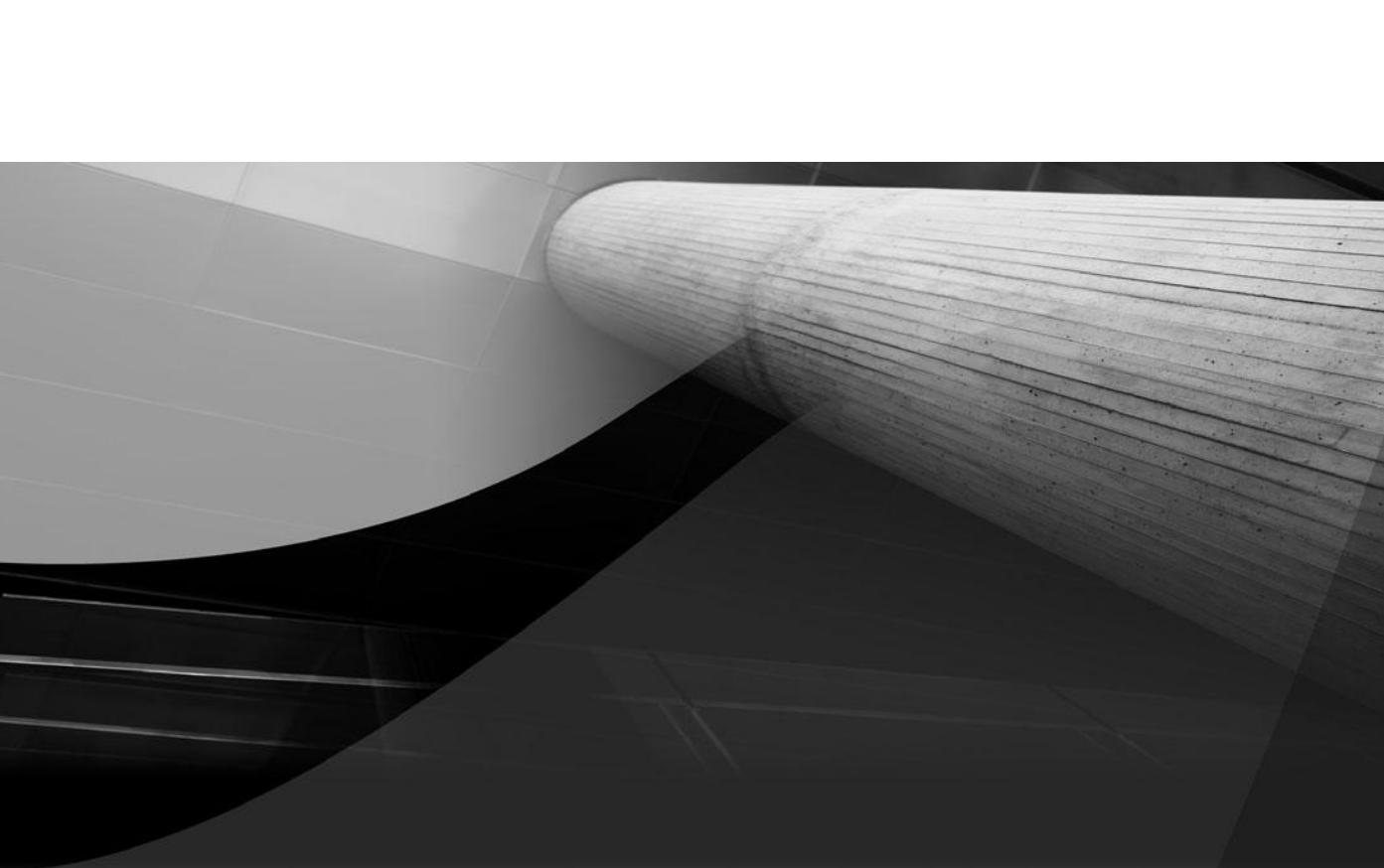
In Chapter 12, you learned about the MediaPlayer and what it is capable of. However, you really only scratched the surface. Therefore Table B-4 lists all the properties available to you in the MediaPlayer.

Property	Type	Description
autoPlay	Boolean	If autoPlay is true, playing will start as soon as possible.
balance	Number	Defines the balance, or left/right setting, of the audio output.
bufferProgressTime	Duration	For a buffered stream, the current buffer position that indicates how much media can be played without stalling the MediaPlayer.
currentCount	Number	Defines the current number of times the media has repeated.

**Table B-4** MediaPlayer Properties

Property	Type	Description
currentTime	Duration	The current media time. May be read to indicate the current position or written to cause the media to seek to the set position.
enabledTracks	Track[]	The sequence of tracks currently enabled on this MediaPlayer.
fader	Number	The fader, or forward and back setting, of audio output on 4+ channel output.
media	Media	Defines the source Media to be played.
mute	Boolean	When true, the player's audio is muted (false otherwise).
onBuffering	function(:Duration):Void	This property is unimplemented and may be deprecated in future releases.
onEndOfMedia	function():Void	Invoked when the player's currentTime reaches stopTime and is not repeating.
onError	function(:MediaError):Void	The onError function is called when a MediaError occurs on this player.
onRepeat	function():Void	Invoked when the player reaches stopTime and will be repeating.
onStalled	function(:Duration):Void	This property is unimplemented and may be deprecated in future releases.
paused	Boolean	Indicates if the player has been paused, either programmatically, by the user, or because the media has finished playing.
rate	Number	Defines the rate at which the media is being played.
repeatCount	Number	Defines the number of times the media should be played.
startTime	Duration	Defines the time offset where media should start playing, or restart from when repeating.
status	Integer	Reflects the current status of the MediaPlayer.
stopTime	Duration	Defines the time offset where media should stop playing or restart when repeating.
supportsMultiViews	Boolean	Indicates if this player can have multiple views associated with it.
timers	MediaTimer[]	The sequence of media timers for this player.
volume	Number	Defines the volume at which the media is being played.

**Table B-4** MediaPlayer Properties (continued)



# Appendix C

## JavaFX Command-Line Arguments

Throughout this book you've relied on the NetBeans IDE to run and compile your JavaFX scripts and applications. This is a convenient way to develop and is by far the most common. Most graphics-based open-system developers choose to use a graphics-based IDE to develop their products.

JavaFX, like most Java-based products, also comes with two powerful command-line utilities that can be used for compiling and executing your scripts. This appendix serves as a reference for some of the capabilities of the JavaFX command-line utilities.

The two main javafx command-line utilities, and their uses, are listed here:

```
javafx      :used to execute javafx applications from the command line  
javafxc     :used to compile javafx scripts from the command line
```

The following sections explain the uses of these utilities. Refer to these sections as needed to help you with your command-line functions. The first section quickly describes the environment you must have in place before you can use the JavaFX command-line utilities.

## Command-Line Environment

If you have followed this book from the beginning and have successfully installed JavaFX, then chances are your machine is already correctly configured for using the command-line utilities. The command-line utilities are installed at the same time as JavaFX. In fact, NetBeans and Eclipse use the same command-line utilities you will be using here.

If you have not followed this book from the beginning and are not sure if your environment is set up to use the command-line utilities, follow these steps:

1. Install JavaFX if you have not already done so. It should go without saying that you need to install JavaFX before you can use any of the JavaFX command-line utilities. Chances are that the installation of JavaFX will take care of setting up your environment variables.
2. Check your Path statement. The JavaFX command-line utilities require that you have a correctly established Path statement. To check it, type **path** in a command window. You should see something similar to this:

```
C:\>path  
PATH=C:\Program Files\JavaFX\javafx-sdk1.3\bin;C:\Program Files  
\\JavaFX\javafx-sdk1.3\emulator\bin;c:\program files\java\jdk1.6.0_13\bin\;
```

The following path is what you are looking for to ensure your environment is set up to run the javafx command-line utilities:

```
C:\Program Files\JavaFX\javafx-sdk1.3\bin;
```

**NOTE**

This path assumes you installed JavaFX in its default location. If you chose a different location in which to install JavaFX, the path should point to that location.

**NOTE**

All the command-line utilities are in the bin folder of the javafx install path.

3. If your path statement does not include the path to the javafx bin folder, you will need to add it. The command to add a path statement looks like this:

```
C:\> set path=C:\Program Files\JavaFX\javafx-sdk1.3\bin; %PATH%
```

**NOTE**

The %PATH% at the end of the set path command tells your system to insert whatever paths are already in your path variable after the new line. If you do not include this, all your existing paths will be replaced by the new one.

With your environment set up correctly, you can now move on to using the command-line utilities.

## javafxc

The javafxc command-line utility is used to compile JavaFX scripts into executable applications. The only required parameter needed to run the javafxc utility is the name of the script file to compile. The following example compiles a file named HelloWorld.fx:

**NOTE**

You can use the HelloWorld file you created in Chapter 3 if you need a file to help you with this example.

```
C:\>javafxc HelloWorld.fx
```

This simple command compiles your script file. Even though only one required parameter is needed to compile a file, you can pass several options into the javafxc utility. These options are listed and explained next.

**Parameter:**

-g

**Explanation:**

Generate all debugging info

**Example:**

```
C:\>javafxc -g HelloWorld.fx
```

**Parameter:**

-g:none

**Explanation:**

Generate no debugging info

**Example:**

```
C:\>javafxc -g:none HelloWorld.fx
```

**Parameter:**

-g:{lines,vars,source}

**Explanation:**

Generate only some debugging info

**Example:**

```
C:\>javafxc -g:{33,34,35,36} HelloWorld.fx
```

**Parameter:**

-nowarn

**Explanation:**

Generate no warnings

**Example:**

```
C:\>javafxc -nowarn HelloWorld.fx
```

**Parameter:**

-verbose

**Explanation:**

Output messages about what the compiler is doing

**Example:**

```
C:\>javafxc -verbose HelloWorld.fx
```

**Parameter:**

-deprecation

**Explanation:**

Output source locations where deprecated APIs are used

**Example:**

```
C:\>javafxc -deprecation HelloWorld.fx
```

**Parameter:**

```
-classpath
```

**Explanation:**

Specify where to find user class files

**Example:**

```
C:\>javafxc -classpath "C:\MyClasses" HelloWorld.fx
```

**Parameter:**

```
-cp
```

**Explanation:**

Specify where to find user class files

**Example:**

```
C:\>javafxc -cp "C:\MyClasses" HelloWorld.fx
```

**Parameter:**

```
-sourcepath
```

**Explanation:**

Specify where to find input source files

**Example:**

```
C:\>javafxc -sourcepath "C:\MySource" HelloWorld.fx
```

**Parameter:**

```
-bootclasspath
```

**Explanation:**

Override the location of bootstrap class files

**Example:**

```
C:\>javafxc -bootclasspath "C:\MyClasses" HelloWorld.fx
```

**Parameter:**

```
-extdirs
```

**Explanation:**

Override the location of installed extensions

**Example:**

```
C:\>javafxc -extdirs "C:\MyExtensions" HelloWorld.fx
```

**Parameter:**

```
-endorseddirs
```

**Explanation:**

Override the location of the endorsed standards path

**Example:**

```
C:\>javafxc -endorseddirs "C:\MyStandards" HelloWorld.fx
```

**Parameter:**

```
-d
```

**Explanation:**

Specify where to place generated class files

**Example:**

```
C:\>javafxc -d "C:\MyFinalBuild" HelloWorld.fx
```

**Parameter:**

```
-implicit:
```

**Explanation:**

Specify whether or not to generate class files for implicitly referenced files

**Example:**

```
C:\>javafxc -implicit:none HelloWorld.fx
```

**Parameter:**

```
-encoding
```

**Explanation:**

Specify character encoding used by source files

**Example:**

```
C:\>javafxc -encoding "UTF-8" HelloWorld.fx
```

**Parameter:**

```
-target
```

**Explanation:**

Generate class files for a specific VM version

**Example:**

```
C:\>javafxc -target "1.5" HelloWorld.fx
```

**Parameter:**

-platform

**Explanation:**

Platform translator plug-in

**Example:**

```
C:\>javafxc -platform "active" HelloWorld.fx
```

**Parameter:**

-version

**Explanation:**

Version information

**Example:**

```
C:\>javafxc -version
```

**Parameter:**

-help

**Explanation:**

Print a synopsis of standard options

**Example:**

```
C:\>javafxc -help
```

**Parameter:**

-X

**Explanation:**

Print a synopsis of nonstandard options

**Example:**

```
C:\>javafxc -X
```

## javafx

The javafx command-line utility is used to execute already compiled JavaFX applications. Like the javafxc utility, javafx can take a number of optional parameters.

### **NOTE**

The javafx command-line utility can execute either a .class file or a .jar file. In the following examples, we use .class.

#### **Parameter:**

-d32

#### **Explanation:**

Use a 32-bit data model if available

#### **Example:**

```
C:\>javafx -d32 HelloWorld
```

#### **Parameter:**

-d64

#### **Explanation:**

Use a 64-bit data model if available

#### **Example:**

```
C:\>javafx -d64 HelloWorld
```

#### **Parameter:**

-client

#### **Explanation:**

Select the “client” VM if available

#### **Example:**

```
C:\>javafx -client HelloWorld
```

#### **Parameter:**

-server

#### **Explanation:**

Select the “server” VM if available

**Example:**

```
C:\>javafx -server HelloWorld
```

**Parameter:**

```
-cp
```

**Explanation:**

Class search path of directories and ZIP/JAR files

**Example:**

```
C:\>javafx -cp "C:\MyClasses" HelloWorld
```

**Parameter:**

```
-classpath
```

**Explanation:**

Class search path of directories and ZIP/JAR files

**Example:**

```
C:\>javafx -classpath "C:\MyClasses" HelloWorld
```

**Parameter:**

```
-D
```

**Explanation:**

Set a system property

**Example:**

```
C:\>javafx -D HelloWorld.property = "value" HelloWorld
```

**Parameter:**

```
-verbose
```

**Explanation:**

Enable verbose output

**Example:**

```
C:\>javafx -verbose HelloWorld
```

**Parameter:**

```
-version
```

**Explanation:**

Print the product version and exit

**Example:**

```
C:\>javafx -version
```

**Parameter:**

```
-version:<value>
```

**Explanation:**

Require the specified JRE version to run

**Example:**

```
C:\>javafx -version:1.3 HelloWorld
```

**Parameter:**

```
-showversion
```

**Explanation:**

Print the product version and continue

**Example:**

```
C:\>javafx -showversion
```

**Parameter:**

```
-jre-restrict-search | -jre-no-restrict-search
```

**Explanation:**

Include/exclude user private JREs in the version search

**Example:**

```
C:\>javafx -jre-restrict-search HelloWorld
```

**Parameter:**

```
-? -help
```

**Explanation:**

Print help options

**Example:**

```
C:\>javafx -?
```

**Parameter:**

```
-X
```

**Explanation:**

Print help on nonstandard options

**Example:**

```
C:\>javafx -X
```

**Parameter:**

-ea

**Explanation:**

Enable assertions with the specified granularity

**Example:**

```
C:\>javafx -ea:com.test.package HelloWorld
```

**Parameter:**

-enableassertions

**Explanation:**

Enable assertions with the specified granularity

**Example:**

```
C:\>javafx -enableassertions:com.test.package HelloWorld
```

**Parameter:**

-da

**Explanation:**

Disable assertions with the specified granularity

**Example:**

```
C:\>javafx -da:com.test.package HelloWorld
```

**Parameter:**

-disableassertions

**Explanation:**

Disable assertions with the specified granularity

**Example:**

```
C:\>javafx -disableassertions:com.test.package HelloWorld
```

**Parameter:**

-esa | -enablesystemassertions

**Explanation:**

Enable system assertions

**Example:**

```
C:\>javafx -esa HelloWorld
```

**Parameter:**

```
-dsa | -disableSystemAssertions
```

**Explanation:**

Disable system assertions

**Example:**

```
C:\>javafx -dsa HelloWorld
```

**Parameter:**

```
-agentlib
```

**Explanation:**

Load the native agent library

**Example:**

```
C:\>javafx -agentlib:hprof HelloWorld
```

**Parameter:**

```
-agentpath
```

**Explanation:**

Load the native agent library by full pathname

**Example:**

```
C:\>javafx -agentpath:C:\MyAgentPath HelloWorld
```

**Parameter:**

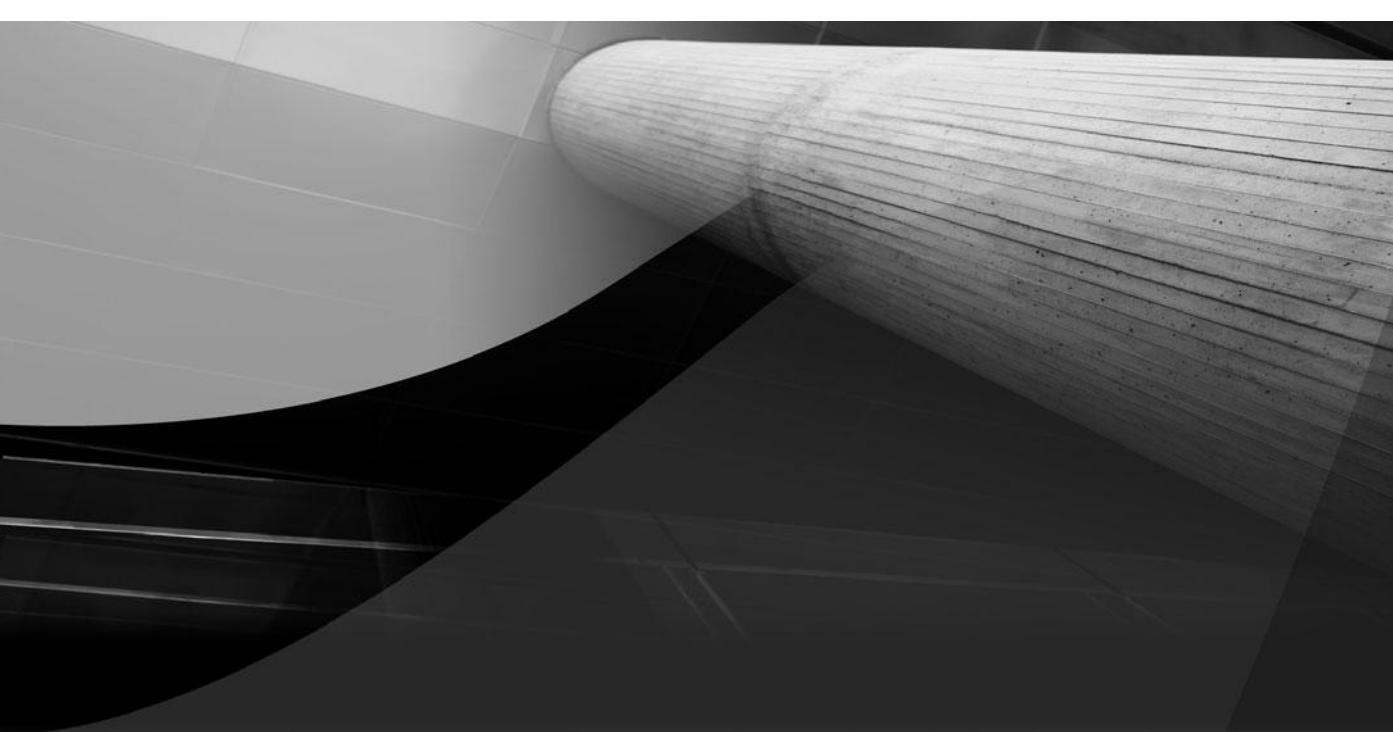
```
-splash
```

**Explanation:**

Show the splash screen with the specified image

**Example:**

```
C:\>javafx -splash:Myslash.png HelloWorld
```



# Appendix D

## Answers to Self Tests

## Chapter 1

- 1. What is the name of the open-source development environment you will use throughout this book?**  
NetBeans
- 2. True or false? You should download the version of NetBeans for All Developers.**  
False. You just need the version for JavaFX Developers.
- 3. True or false? The Java JDK will be installed for you automatically if needed (if you have the JRE installed).**  
True
- 4. Which NetBeans settings can you accept the default values for during installation?**  
The default NetBeans IDE path and the Java JDK installation path
- 5. What is the difference between the JavaFX SDK and the Java JDK?**  
The Java JDK is used to develop and compile in Java. The JavaFX SDK is based on the Java JDK and is used for JavaFX development.
- 6. What is the purpose of the NetBeans start page?**  
The purpose of the NetBeans start page is to offer you tips and news about developing in NetBeans and JavaFX.
- 7. True or false? You must successfully register NetBeans before using it.**  
False
- 8. At what website is NetBeans available?**  
[www.netbeans.org](http://www.netbeans.org)
- 9. Name two other applications that closely resemble the functionality of JavaFX.**  
Adobe Flash and Microsoft Silverlight
- 10. JavaFX will compile for the Desktop, Web, and what other platforms?**  
Mobile and television

## Chapter 2

- 1. What is the name of the frame where all your projects are listed?**  
Projects
- 2. What is the name of the wizard used to create a new JavaFX project?**  
The New JavaFX Project Wizard

**3. What is another name for a namespace?**

A package

**4. Which panel of the NetBeans IDE lets you navigate through code samples?**

The Palette

**5. True or false? The Snippets panel contains predefine pieces of reusable code.**

False. It is the Palette that contains predefine pieces of reusable code.

**6. What file extension is assigned to JavaFX Script files?**

.fx

**7. What type of word is “package” in the JavaFX script?**

A keyword, or reserved word

**8. True or false? Every line of your script must end with a period.**

False. Code lines must end with a semicolon.

**9. What are the beginning and ending characters for comments?**

/\* and \*/

**10. What type of variable or attribute is the following?**

title: "MyApp"

A name-value pair

## Chapter 3

**1. What are the four basic attributes that need to be defined for a Text node?**

font (size), x position, y position, and content

**2. In what Palette menu is the Text node located?**

Basic Shapes

**3. What Run configuration is used to run your script as a Desktop application?**

<default>

**4. When creating a function, where would you specify input parameters?**

Within the parentheses, following the function name

**5. True or false? The function name MyFunction follows proper naming convention.**

False. The name should be camel-cased and use descriptive action words.

**6. What keyword is used to return a value from a function?**

Return.

- 7. Name the two keywords that can be used to create variables.**  
var and def
- 8. How would you type a variable as string?**  
var <variable name> :String
- 9. True or false? The bind keyword is used to bind a variable and make sure it never changes.**  
False. Bind is used to bind a value to a variable.
- 10. What syntax would be used to bind a variable named tooMuchText to a content attribute?**  
content: bind tooMuchText

## Chapter 4

- 1. What four attributes are needed to draw a line?**  
startX, startY, endX, and endY
- 2. How do you access the context menu?**  
CTRL-SPACE
- 3. What three delimiters can follow an attribute definition?**  
comma, semicolon, and nothing
- 4. What attribute controls the thickness of the line used to draw a shape?**  
strokeWidth
- 5. What package is needed to draw a polyline?**  
javafx.scene.shape.Polyline
- 6. What type of value is assigned to the points attribute of a Polyline element?**  
An array
- 7. True or false? The height attribute of the Rectangle element is the number of pixels from the start point to the top of the rectangle.**  
False. The height attribute is the number of pixels from the start point *down*.
- 8. What is the default value for the fill attribute of a Rectangle element?**  
Color.BLACK
- 9. True or false? RadiusX and radiusY comprise the point where the radius extends to.**  
False. RadiusX and radiusY are the radial lengths along the x and y axes, respectively.
- 10. What attribute configures the radius of a circle?**  
radius

## Chapter 5

- 1. How many predefined colors are available in the Color class?**  
148
- 2. What are the three methods available in the Color class for mixing colors?**  
Color.rgb, Color.hsb, and Color.web
- 3. True or false? RGB stands for refraction, gradient, and blur.**  
False. RGB stands for red, green, and blue.
- 4. What is the acceptable value range for Hue?**  
0–360
- 5. In what package is the code for LinearGradients?**  
javafx.scene.paint.LinearGradient
- 6. What is the default value for the proportional parameter?**  
True
- 7. What is the acceptable value for startX when proportional is set to true?**  
0–1
- 8. True or false? The stops parameter tells the gradient what point to stop on.**  
False. The stops parameter is an array of colors and the corresponding indications of where they are in the gradient.
- 9. True or false? Gradients can be composed of more than two colors.**  
True
- 10. Which gradient is best for curvilinear shapes?**  
RadialGradients

## Chapter 6

- 1. What node is used to display images?**  
ImageView
- 2. What class is used to write an image to the ImageView node?**  
Image()
- 3. True or false? An Image class can accept images from the Web.**  
True
- 4. What value does the {\_\_DIR\_\_} constant contain?**  
The path to the package location

- 5. True or false? To have an image load in the background, use the `BackgroundImage` loader.**  
False. Set the `backgroundLoading` attribute to true.
- 6. What is the name of the tool used to export images from Adobe Photoshop and Adobe Illustrator for JavaFX?**  
JavaFX Production Suite
- 7. True or false? You must add `jfx:` to the beginning of each layer name to access those layers by name in your script.**  
True
- 8. What node is used to load images from an FXZ file?**  
`FXDNode`
- 9. True or false? The FXZ file is a compressed file that contains images and definitions.**  
True
- 10. What method is used to load an image layer?**  
`getNode()`

## Chapter 7

- 1. How do you assign a type to a var?**  
Use the `: <type>` notation. For example, use `: ImageView` to type a var as an `ImageView`.
- 2. What effect adjusts only the higher contrast areas of your node to make them glow?**  
`Bloom`
- 3. True or false? All the parameters of `ColorAdjust` default to 0 if they are not specified.**  
False. Contrast defaults to 1.
- 4. What parameter needs to be specified to create a `GaussianBlur` effect?**  
`radius`
- 5. What is the difference between Glow and Bloom?**  
Glow is applied to the entire image, whereas Bloom only applies to the areas of higher contrast.
- 6. True or false? You do not need to specify both a radius and a height/width for a `DropShadow`.**  
True

- 7. Which effect takes all the opaque areas of your image and makes them transparent?**

InvertMask

- 8. What are the three different lights that can be used in the Lighting Effect?**

DistantLight, PointLight, and SpotLight

- 9. What does the following code do?**

```
butterfly.rotate = 45;
```

It rotates the butterfly image 45 degrees.

- 10. How many parameters need to be set to create a PerspectiveTransform effect?**

Eight

## Chapter 8

- 1. Why is timing important to animation?**

Timing is critical to producing smooth animation.

- 2. What controls the timer in JavaFX animation?**

A Timeline

- 3. What does a Timeline take in as a parameter?**

keyFrame

- 4. How do you start the Timeline?**

.play()

- 5. True or false? The transition notation tells the Timeline to build all the values between the ones specified in your keyframes.**

False. It is the keyword tween that does this.

- 6. Which parameter sets the number of times a Timeline executes?**

repeatCount

- 7. What is the purpose of ClosePath()?**

To “connect the dots” and close your path (that is, if it does not close organically)

- 8. A path is created from a group of what?**

Elements

- 9. What function is used to create an AnimationPath from a Path node?**

createFromPath()

- 10. Which OrientationType will change the orientation of the node as it moves along the path?**

ORTHOGONAL\_TO\_TANGENT

## Chapter 9

- 1. Where are the onMouse\* events inherited from?**

Node

- 2. When is onMouseEntered fired?**

When the mouse pointer enters the Node to which the event is attached

- 3. True or false? The onMouseReleased event only fires when the mouse is dragged.**

True. onMouseReleased is only fired when onMousePressed is followed by onMouseDragged.

- 4. True or false? Anything that inherits from Node can trap onMouse\* events.**

True

- 5. When events are used, what is the purpose of an anonymous function?**

The purpose of an anonymous function is to immediately perform an action when the event is fired.

- 6. Which mouse event is fired when the mouse pointer exits the node to which the event is attached?**

onMouseExited

- 7. What three events are fired when the user interacts with the keyboard?**

onKeyPressed, onKeyReleased, and onKeyTyped

- 8. In what order are the key events fired?**

onKeyPressed, onKeyTyped, and then onKeyReleased

- 9. What property will allow a node to accept focus?**

focusTraversable

- 10. True or false? The navigational buttons on a mobile phone will fire the onKeyTyped event.**

False

## Chapter 10

- 1. What package contains the swing components for JavaFX?**

javafx.ext.swing

- 2. True or false? The swing JavaFX package contains all the components available in Java.**

False. The JavaFX swing package only contains a subset of what is available in Java.

**3. Are the onMouse\* and onKey\* events available to swing components?**

Yes, swing components for JavaFX inherit from Node.

**4. What are the JavaFX string interpolator operators?**

{ and }

**5. What property of the SwingButton can hold an anonymous function that will execute when the button is clicked?**

action

**6. What property of the SwingButton can be used to change the shape of the button?**

clip

**7. True or false? The isChecked property of the SwingCheckBox will tell you if the box is checked.**

False. The selected property will tell you if the box is checked.

**8. What swing component is used to populate SwingComboBox?**

SwingComboBoxItem

**9. How do you set a SwingComboBoxItem to the default choice?**

Set its selected property to true.

**10. What property of SwingComboBox will tell you what SwingComboBoxItem has been selected?**

selectedItem

## Chapter 11

**1. What process lets you take methods and properties from one class and change their default actions and behaviors?**

Overriding

**2. When you're creating a class, what keyword forces your class to inherit the methods and properties of another?**

extends

**3. In the following example, what will a call to YourDog.displayBreed print?**

```
public class MyDog extends Dog{  
    override function displayBreed() {  
        println("Elkhound");  
    }  
}  
public class YourDog extends Dog{  
}
```

Elkhound

4. True or false? Ensuring that your files are all in the same package will make referencing them easier.  
True
5. True or false? After inheriting from a class, only attributes that you override are available to you in another class.  
False. All methods and attributes are available to you.
6. What trigger will execute when an attribute changes?  
on replace
7. What statement will take an expression for a true or false result and then execute code accordingly?  
if...else
8. True or false? You have to call a custom-created node from a script to use it.  
True
9. What node do you inherit from to create a custom node?  
CustomNode
10. What method of CustomNode do you override to return your node to the calling script?  
create()

## Chapter 12

1. What node is used to hold a MediaPlayer?  
MediaView
2. What package contains all the nodes needed to work with media files?  
javafx.scene.media
3. What property of the MediaPlayer tells the media file to play once it has loaded?  
autoPlay
4. What media formats can the MediaPlayer play?  
Any format supported by QuickTime or Windows Media Player
5. What property of the MediaPlayer will pause media playback?  
pause()
6. True or false? MediaPlayer.mediaLength() will give you the total running time of a media file.  
False. MediaPlayer.media.duration.toMillis() will give you the running time in milliseconds.

**7. What type is MediaPlayer.currentTime?**

Duration

**8. What type of binding allows for bidirectional updating?**

binding with inverse

**9. True or false? Using inverse binding, you can bind directly to a value.**

False. You must bind indirectly through a variable.

**10. What property of MediaPlayer can you bind to in controlling the playback volume?**

volume

## Chapter 13

**1. What layout organizes your nodes horizontally across a Scene?**

HBox

**2. True or false? The HBox is located in the javafx.scene.HBox package.**

False. The HBox is located in the javafx.scene.layout package.

**3. What property holds the nodes for a layout to organize?**

content

**4. True or false? You must be sure to set the x and y coordinates of each node you place in a layout.**

False. The layout takes care of the x and y coordinates for you.

**5. Can effects be applied to layouts?**

Yes. Because layouts inherit from Node, they can use effects.

**6. What layout organizes nodes vertically down a Scene?**

VBox

**7. What is the name given to layouts that are combined to produce a new layout?**

Nested layouts

**8. True or false? For layouts to be nested, one must inherit from the other.**

False. One layout simply needs to be added to the other's content.

**9. True or false? Only two layouts can be nested.**

False. Multiple layouts can be nested.

**10. Name four layouts other than the VBox and HBox.**

ClipView, Flow, Stack, and Tile

## Chapter 14

### 1. What is Cascading Style Sheets (CSS)?

CSS is a styling language that allows you to separate the styling elements of an object from the object itself.

### 2. What file extension is used for Cascading Style Sheets?

.css

### 3. What wizard helps you create and add a CSS to your package?

The New File Wizard

### 4. If you use the wizard to create your CSS, what class is added by default?

root

### 5. True or false? To create a CSS class that applies to all nodes of a certain type, the name of the class should be the name of the node type in lowercase.

True

### 6. What prefix is added to every Node property to call it from a CSS class?

-fx-

### 7. What property of Scene will let you apply a style sheet to your script?

styleSheets

### 8. True or false? You can only add one style sheet to a Scene.

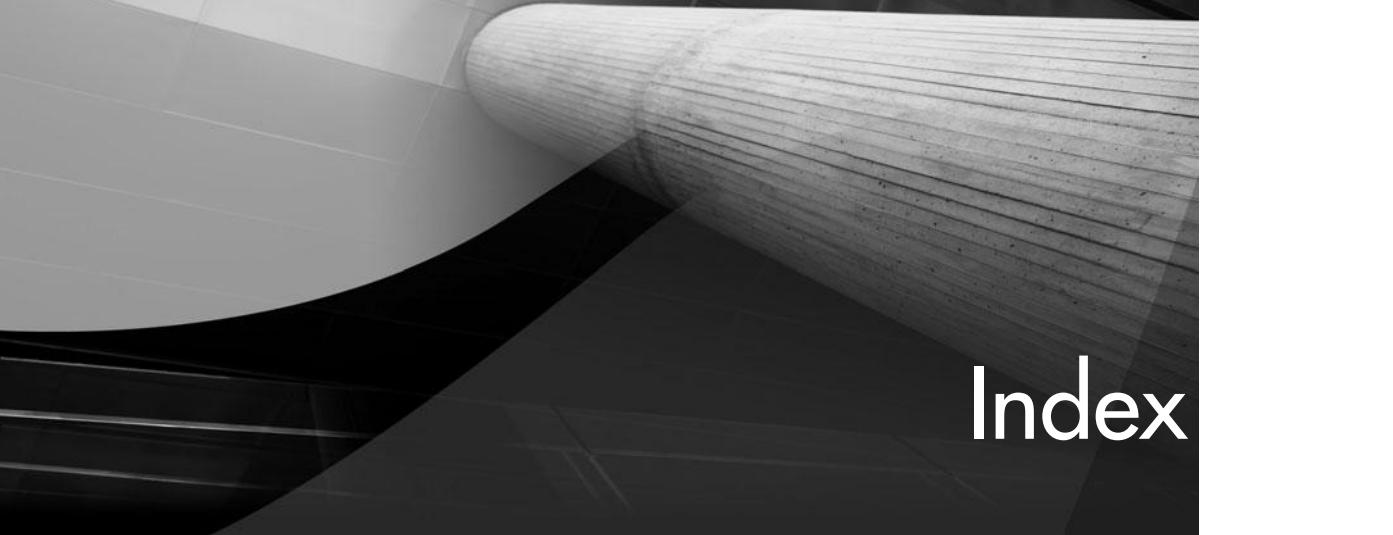
False. You can add multiple style sheets to any particular Scene.

### 9. What Node property allows you to assign a specific CSS class to a node?

styleClass

### 10. True or false? If you have a node with a node-applied CSS class and a styleClass property, the style in the styleClass will override that in the node-applied style.

True



# Index

## A

---

action property, 168  
Adobe Photoshop, 91–93  
-agentlib parameter, 278  
-agentpath parameter, 278  
alpha attribute, 76–77  
altDown event, 257  
animation  
    overview, 132  
    along paths, 139–145  
    timelines, 133–139  
AnimationPath class, 142  
applets vs. applications, 38  
applications  
    vs. applets, 38  
    compiling, 269–273  
    with swing, 180  
Applications tab, 21  
ApplyEffects function, 103–104  
    Bloom effect, 106–108  
    ColorAdjust effect, 110  
    DistantLights, 120–121  
    DropShadow effect, 116–117  
    GaussianBlur effect, 111  
    Glow effect, 114–115

InvertMask effect, 119  
PerspectiveTransform, 128  
PointLights, 122  
SepiaTone effect, 125  
SpotLights, 123  
Arc attribute, 67  
arcHeight attribute, 63–64  
arcs, drawing, 65–68  
ArcTo function, 141  
ArcType attribute, 67  
arcWidth attribute, 63–64  
arrays, 60  
asterisks (\*) for comments, 20  
attributes  
    fonts, 32  
    name-value pairs, 23  
    Palette, 33  
audio files  
    overview, 201–202  
    playing, 216–217  
autoPlay property, 204, 207–208, 265  
autoReverse parameter  
    PathTransition, 142  
    Timeline, 138  
azimuth parameter, 120

**B**

backgroundLoading attribute, 89  
balance property, 265  
Basic Shapes section, 32  
bin folder, 269  
bind keyword, 46  
binding  
  inverse, 213–214  
  string interpolators, 174  
  Text node, 46–49  
black-and-white film effect, 123–126  
blocksMouse property, 254  
Bloom effect, 106–108  
blurring effects, 110–113  
blurType parameter, 116  
-bootclasspath parameter, 271  
boundsIn property, 254  
brightness attribute  
  ColorAdjust effect, 109  
  HSB color, 76  
bufferProgressTime property, 265  
Build Project option, 248–249  
button event, 257  
buttons  
  horizontal layout, 221–225  
  nested layouts, 227–230  
  play/pause, 207–211  
  RoundButton, 186–192  
  SwipeButton, 166–172  
  vertical layout, 225–227

**C**

cache property, 254  
camel casing, 41  
Cartesian coordinates  
  in animation, 141  
  axes, 33  
Cascading Style Sheets (CSS), 234–235  
  independent style classes, 241–244  
  packages, 235–238  
  style creation, 238–239  
  working with, 239–241  
center points for RadialGradients, 81  
centerX attribute, 68  
centerY attribute, 68  
check boxes, 173–176  
chord attribute, 67

circles, 66  
  drawing, 68–69  
  RadialGradients, 81  
classes  
  extending, 184  
  in packages, 12–13, 204  
-classpath parameter  
   javafx, 275  
  javafxc, 271  
clickCount event, 257  
-client parameter, 274  
clip property  
  description, 254  
  overriding, 186  
  SwipeButton, 170–172  
ClosePath function, 141  
code snippets  
  categories, 18  
  need for, 25  
  Stage node, 21–23  
colons (:) in attribute values, 55  
color, 72  
  fonts, 238  
  LinearGradient, 78  
  lines, 57  
  mixing, 74–77  
  polygons, 64  
  predefined, 72–74  
  RadialGradients, 81  
  rectangles, 63  
Color class, 72, 74–77  
color parameter  
  DistantLights, 120  
  DropShadow effect, 116  
ColorAdjust effect, 109–110  
combining effects, 130  
combo boxes, 176–180  
command-line environment  
  javafx, 274–278  
  javafxc, 269–273  
  setting up, 268–269  
commas (,)  
  array values, 60  
  attributes, 55  
  parameters, 42  
comments  
  adding, 30  
  CSS, 238  
  overview, 19–21

compiling  
     process, 21  
     scripts, 24, 269–273  
 content.fxd file, 95–96  
 content property, 223  
 context menus, 14, 54–56  
 contrast parameter, 109–110  
 controlDown event, 257  
 coordinates  
     Cartesian, 33  
     PerspectiveTransform, 128–129  
 copyright information in comments, 19  
 -cp parameter  
     javafx, 275  
     javafxc, 271  
 Create File dialog box, 90  
 create method for nodes, 193–197  
 createFromPath function, 142  
 CSS (Cascading Style Sheets), 234–235  
     independent style classes, 241–244  
     packages, 235–238  
     style creation, 238–239  
     working with, 239–241  
 curly braces ({} )  
     for context menus, 54–55  
     functions, 41–42  
     string interpolation, 168  
     type names, 23  
 currentCount property, 265  
 currentTime property, 211–212, 266  
 cursor property, 254  
 custom nodes, 192–199

**D**

-D parameter, 275  
 -d parameter, 272  
 -d32 parameter, 274  
 -d64 parameter, 274  
 -da parameter, 277  
 dates in comments, 19–20  
 Debug option, 31  
 def keyword, 46  
 default.css file, 237, 241  
 default package names, 15  
 definitions, function, 104  
 delimiters, 21  
 deploying JavaFX, 248–251

-deprecation parameter, 270–271  
 descriptive comments, 30  
 desktop profiles, 35–36  
 {\_\_DIR\_\_} constant, 89–91, 136  
 disable property, 254  
 -disableassertions parameter, 277  
 disabled property, 254  
 -disablesystemassertions parameter, 278  
 dist folder, 248, 250  
 DistantLights, 120–121  
 distortion, GaussianBlur effect for, 112  
 distributed apps, 251  
 distributing images, 90  
 .docx files, 15  
 double plus sign operator (++) for incrementing, 169  
 downloading software, 4–7  
 drag-and-drop interface, 3  
 drag events, 257  
 DropShadow effect, 116–118  
 -dsa parameter, 278  
 duration property  
     PathTransition, 142  
     progress indicator, 211–212

**E**


---

-ea parameter, 277  
 Eclipse IDE, 7  
 effect property, 255  
 effects  
     Bloom, 106–108  
     ColorAdjust, 109–110  
     combining, 130  
     description, 106  
     DropShadow, 116–118  
     GaussianBlur, 110–113  
     Glow, 113–115  
     InvertMask, 119  
     Lighting, 120–123  
     script setup for, 103–106  
     SepiaTone, 123–126  
     Shadow, 118  
 elevation parameter, 120  
 ellipses  
     drawing, 69  
     RadialGradients, 81  
 -enableassertions parameter, 277  
 enabledTracks property, 266

-enablesystemassertions parameter, 277  
 -encoding parameter, 272  
 -endorseddirs parameter, 272  
 endX value  
     LinearGradient, 78  
     lines, 54, 56  
 endY value  
     LinearGradient, 78  
     lines, 54, 56  
 -esa parameter, 277–278  
 events  
     description, 147  
     key, 157–161  
     in layouts, 224  
     mouse, 148–157, 256–257

Excel files, 15  
 executable applications, compiling, 269–273  
 exporting images, 93–95  
 -extdirs parameter, 271–272  
 extending classes, 184  
 extracting layers, 97

**F**

fader property, 266  
 Files tab, 16–17  
 fill attribute  
     arcs, 67  
     rectangles, 62  
 focusable property, 255  
 focused property, 255  
 focusTraversable property, 159  
 folders for images, 90  
 Font class, 32–33  
 fonts  
     labels, 238–239  
     text, 32–33  
 full transparency, 77  
 fully opaque color, 77  
 function keyword, 41  
 functions  
     adding, 40–46  
     definitions, 104  
 .fx files, 15  
 FXDNode, 96–97  
 FXZ files  
     creating, 93  
     effects, 102, 104  
     layered images, 91, 95  
     working with, 96–98

**G**


---

-g parameter, 269–270  
 Gaussian algorithm, 110  
 GaussianBlur effect, 110–113  
 getNode method, 97, 136  
 Glow effect, 113–115  
 gradients, 77  
     custom, 82  
     LinearGradients, 77–81  
     RadialGradients, 81–82  
 graphic buttons, 166–172

**H**


---

half transparency, 77  
 HBox layout, 221–225  
 height attribute  
     arcs, 67  
     DropShadow effect, 116  
     images, 88  
     rectangles, 62  
 Hello World script, 28  
     bind for, 46–49  
     comments, 30  
     functions, 40–46  
     Run configuration, 40  
     Stage and Scene, 30–31  
     text, 31–38  
     writing to screen, 28–29  
 -help parameter  
     javafx, 276  
     javafxc, 273  
 horizontal layout, 221–225  
 hover property, 255  
 HTML knowledge requirements, 3  
 Hue, Saturation, Brightness (HSB) color, 76  
 hue parameter for ColorAdjust, 109  
 hyphens (-) for properties, 238

**I**


---

id property, 255  
 IDE (Integrated Development Environment), 4–7  
 Image class, 87–91  
 images, 86  
     exporting, 93–95  
     FXZ files, 96–98  
     Image class, 87–91

ImageView node, 86–87  
 JavaFX Production Suite, 91–96  
 resizing, 112  
 rotating, 127–128  
 types, 99  
**ImageView node**  
 importing, 86–87  
 rotating, 127–128  
**-implicit parameter**, 272  
**import keyword**  
 nodes, 34–35  
 packages, 22  
**independent style classes**, 241–244  
**inference variable type**, 47  
**inheriting mouse events**, 151, 157  
**Insert Template: Text Wizard**, 34  
**installing software**, 4–7  
**Integrated Development Environment (IDE)**, 4–7  
**interpolators**  
 animation, 142  
 strings, 168, 174, 177–178  
**inverse binding**, 213–214  
**InvertMask effect**, 119  
**items property**, 177

**J**

**Java Archive (JAR) files**, 12–13  
**Java development knowledge requirements**, 3  
**Java SE JDK (Standard Edition Java Development Kit)**, 4  
**javafx.animation.Interpolator package**, 137  
**javafx.animation.Timeline package**, 133  
**javafx.awt package**, 153  
**javafx command-line utility**, 274–278  
**javafx.ext.swing package**, 164–165  
**javafx.ext.swing.SwingButton package**, 168, 186  
**javafx.ext.swing.SwingCheckBox package**, 174  
**javafx.ext.swing.SwingComboBox package**, 177  
**javafx.ext.swing.SwingTextField package**, 194  
**javafx.fxd.FXDNNode package**, 96  
**JavaFX Production Suite**, 91–96  
**javafx.scene.control.Label package**, 240  
**javafx.scene.control.Slider package**, 214  
**javafx.scene.effect package**, 106  
**javafx.scene.Group package**, 194  
**javafx.scene.image.Image package**, 87  
**javafx.scene.image.ImageView package**, 86  
**javafx.scene.input.MouseEvent package**, 153  
**javafx.scene.layout package**, 222, 225

**javafx.scene.media.Media package**, 205  
**javafx.scene.media.MediaPlayer package**, 204  
**javafx.scene.paint.Color package**, 57, 67, 72  
**javafx.scene.paint.LinearGradient package**, 77  
**javafx.scene.paint.RadialGradient package**, 81  
**javafx.scene.Scene package**, 136  
**javafx.scene.shape.Arc package**, 66  
**javafx.scene.shape.ArcType package**, 66–67  
**javafx.scene.shape.Circle package**, 68, 81, 186  
**javafx.scene.shape.Color package**, 68  
**javafx.scene.shape.Path package**, 141  
**javafx.scene.shape.Polyline package**, 59  
**javafx.scene.shape.Rectangle package**, 61  
**JavaFX Script language**, 3  
**JavaFX SDK (Software Development Kit)**, 4  
**javafxc command-line utility**, 269–273  
**jfx prefix**, 92  
**-jre-restrict-search parameter**, 276

**K**


---

**key codes**, 258–265  
**key events**, 157–161  
**KeyEvent class**, 161  
**keyframes**  
 animation, 136–137  
 description, 133  
**keyFrames collections**, 136  
**keywords**, 20–21  
**knowledge requirements**, 3

**L**


---

**labels**, 238–239  
**layers**  
 extracting, 97  
 JavaFX Production Suite, 92–93  
**layoutBounds property**, 255  
**layouts**, 220–221  
 HBox, 221–225  
 miscellaneous, 230  
 nested, 227–230  
 VBox, 225–227  
**level parameter**, 113  
**light parameter**, 120  
**Lighting effect**, 120  
 DistantLights, 120–121  
 PointLights, 121–122  
 SpotLights, 123

LinearGradient class, 77–81  
 lines, 53–59  
 local image files, 88, 90  
 location of projects, 10, 12

**M**

Main class, 29  
 media property, 204, 211, 266  
 MediaPlayer, 202–203  
     properties, 205–206, 265–266  
     video, 203–204  
 MediaView node, 202–203  
 metaDown event, 257  
 middleButtonDown event, 257  
 milliseconds for video duration, 211  
 mixing colors, 74–77  
 Mobile profile, 159–160  
 mouse events, 148–157, 256–257  
 MouseEvent class, 153  
 MoveTo function, 141  
 MP3 files, 216–217  
 multiline comments, 19, 21  
 multiple effects, 130  
 multiple shapes, 70  
 music, 216–217  
 mute property, 266  
 MyMediaPlayer class, 204

**N**

name-value pairs, 23–24  
 names  
     in comments, 19  
     conventions, 14  
     functions, 41, 104  
     layers, 92–93  
     nodes, 23  
     packages, 13–15, 21  
     projects, 10, 12  
     variables, 46  
 nested layouts, 227–230  
 NetBeans development environment  
     configuring, 7  
     description, 4  
     downloading and installing, 4–7  
     empty projects, 15–18  
 New File Wizard, 235, 238  
 New Project Wizard, 10–12  
 node event, 257

nodes  
     creating, 192–199  
     importing for, 34–35  
     overriding, 184–185  
     properties, 238, 254–256  
     rotating, 241–244  
     type names, 23  
 Notes class, 193–199  
 -nowarn parameter, 270

**O**

octagons, 65  
 offset parameter, 78–80  
 on replace triggers, 187  
 onBuffering property, 266  
 onEndOfMedia property, 266  
 onError property, 266  
 onKey events in layouts, 224  
 onKey properties, 255  
 onKeyPressed event, 157–158, 161  
 onKeyReleased event, 157–158, 161  
 onKeyTyped event, 158, 161  
 onMouse events in layouts, 224  
 onMouse properties, 255  
 onMouseClicked event, 149–151, 153, 155  
 onMouseDragged event, 150  
 onMouseEntered event, 150  
 onMouseExited event, 150  
 onMouseMoved event, 150  
 onMousePressed event, 149–151, 154  
 onMouseReleased event, 149–151, 153  
 onMouseWheelMoved event, 148, 150  
 onRepeat property, 266  
 onStalled property, 266  
 opacity  
     alpha attribute, 76–77  
     DropShadow effect, 116  
     InvertMask effect, 119  
 opacity property, 256  
 open attribute, 67  
 orientation of animation, 143  
 overriding nodes, 184–185

**P**

package statement, 20–21  
 packages  
     adding images to, 90  
     naming conventions, 13–15

- overview, 12–13
  - style sheets in, 235–238
- Palette**
  - attributes, 33
  - NetBeans, 16–18
  - for Stage, 30
  - for text, 32
- parameters**
  - functions, 41–42
  - name-value pairs, 23
- parent property**, 256
- parentheses () for parameters**, 42
- Path node**, 140–141
- Path statement**, 268–269
- paths**
  - animation along, 139–145
  - command-line environment, 268–269
- PathTransition class**, 142
- pause button**, 207–211
- pause method**, 209
- paused property**, 266
- PerspectiveTransform transformations**, 128–129
- Photoshop**, 91–93
- placeholder images**, 89
- platform parameter**, 273
- play method**, 136, 209
- play/pause button**, 207–211
- playing**
  - audio, 216–217
  - video, 203–207
- plus signs (+) in increment operator**, 169
- pointer-style input events**, 148–157
- PointLights**, 121–122
- points attribute**, 60
- pointsAtX parameter**, 123
- pointsAtY parameter**, 123
- pointsAtZ parameter**, 123
- polygons**, 64–65
- Polyline package**, 59
- polylines**, 59–61
- popupTrigger event**, 257
- position, layouts for**, 220–221
  - HBox, 221–225
  - miscellaneous, 230
  - nested, 227–230
  - VBox, 225–227
- predefined colors**, 72–74
- pressed property**, 256
- primaryButtonDown event**, 257
- {\_\_PROFILE\_\_} constant**, 35, 37, 44
- profiles**
  - desktop, 35–36
  - Mobile, 159–160
- progress indicators**, 211–215
- Project context menu**, 14
- Project Properties window**, 36–37
- projects**
  - creating, 10–12
  - empty, 12–15
  - NetBeans, 15–18
  - properties, 29, 36–37
  - working files, 15
- Projects frame**, 12, 29
- Projects tab in NetBeans**, 16–17
- properties**
  - Node, 254–256
  - projects, 29, 36–37
- Properties dialog box**, 248
- proportional parameter**, 78

---

**Q**

- 
- question mark (-?) parameter, 276

---

**R**

- 
- RadialGradients**, 81–82
  - radius**
    - arcs, 66
    - circles, 68
    - DropShadow effect, 116
    - GaussianBlur effect, 110–113
    - RadialGradients, 81
  - radiusX attribute**, 66
  - radiusY attribute**, 66
  - rate property**, 266
  - rectangles**, 61–64
  - Red, Green, Blue (RGB) color**, 75–76
  - registering NetBeans**, 7
  - renaming layers**, 92–93
  - repeatCount property**
    - animation, 138–139
    - description, 266
    - PathTransition, 142
  - requirements for development**, 2–3
    - skills and knowledge, 3
    - software, 4
  - reserved words**, 20–21
  - resizing images**, 112

return keyword, 44  
 return value types, 42–43  
 RGB (Red, Green, Blue) color, 75–76  
 rotate property, 241–244, 256  
 rotating  
     images, 127–128  
     nodes, 241–244  
 round attribute, 67  
 RoundButton  
     creating, 186–192  
     play/pause button, 207–211  
 Run configurations  
     for compilation, 24  
     creating, 40  
 Run Main Project option, 31  
 Run properties, 248–249

**S**

saturation attribute  
     ColorAdjust, 109  
     HSB color, 76  
 saving images, 93–94  
 sayHello function, 43–45, 47  
 sayHelloFromBind function, 47–49  
 scaleX property, 256  
 scaleY property, 256  
 Scene node, 31  
 scene property, 256  
 Scenes, adding, 30–31  
 sceneX event, 257  
 sceneY event, 257  
 screen, writing to, 28–29  
 screenX event, 257  
 screenY event, 257  
 script files, 19  
     comments, 19–21  
     compiling, 24, 269–273  
     Hello World. *See* Hello World script  
     name-value pairs, 23–24  
     package statement, 20–21  
     Stage node, 21–23  
 secondaryButtonDown event, 257  
 selected property, 178  
 self-test answers, 280–290  
 semicolons (:)  
     attributes, 55  
     functions, 42  
     statements, 21

SepiaTone effect, 123–126  
 -server parameter, 274  
 Services tab, 16–17  
 SetImages function, 103–105  
     Bloom effect, 106–108  
     DropShadow effect, 117  
     Glow effect, 114–115  
     rotation transformation, 127–128  
     SepiaTone effect, 125  
     XY transformations, 125–126  
 shadows  
     DropShadow effect, 116–118  
     Shadow effect, 118  
 shapes, 52  
     arcs, 65–68  
     circles, 68–69  
     creating, 199  
     drawing, 52–53  
     ellipses, 69  
     lines, 53–59  
     multiple, 70  
     Palette, 32  
     polygons, 64–65  
     polylines, 59–61  
     rectangles, 61–64  
 shiftDown event, 257  
 Show On Startup option, 7  
 -showversion parameter, 276  
 single line comments, 19, 21  
 size  
     fonts, 32  
     images, 112  
 skills requirements, 3  
 slashes (/) for comments, 20  
 Slider control, 211, 213–214  
 snippets  
     categories, 18  
     need for, 25  
     Stage node, 21–23  
 software  
     downloading and installing, 4–7  
     requirements, 4  
 Software Development Kit (JavaFX SDK), 4  
 sound files  
     overview, 201–202  
     playing, 216–217  
 Source Packages folder, 12, 14  
 -sourcepath parameter, 271  
 -splash parameter, 278  
 SpotLights, 123

- spread parameter, 116
  - square brackets ([])
    - arrays, 60
    - Text node, 32
  - squares, 61
  - Stage node
    - adding, 30–31
    - name-value pairs, 23–24
    - snippets, 21–23
  - Standard Edition Java Development Kit (Java SE JDK), 4
  - start page in NetBeans, 6
  - startAngle property, 66
  - startTime property, 266
  - startX value
    - LinearGradient, 78
    - lines, 54
  - startY value
    - LinearGradient, 78
    - lines, 54
  - status property, 266
  - stops parameter
    - LinearGradient, 78–79
    - RadialGradient, 81
  - stopTime property, 266
  - streaming media, 202
  - strings
    - with functions, 43–44
    - interpolation, 168, 174, 177–178
  - stroke attribute
    - arcs, 67
    - lines, 57
    - rectangles, 62–63
  - strokeWidth attribute
    - arcs, 67
    - rectangles, 62–63
  - style property, 234, 256
  - styleClass property, 242–244, 256
  - styles, CSS, 234–235
    - independent style classes, 241–244
    - packages, 235–238
    - style creation, 238–239
    - working with, 239–241
  - styleSheets property, 240
  - supportsMultiViews property, 266
  - swing
    - applications with, 180
    - components, 165–166
    - overview, 164–165
    - SwingButton, 166–172
  - SwingCheckBox, 173–176
  - SwingComboBox and SwingComboBoxItem, 176–180
  - SwingButton
    - creating, 166–172
    - RoundButton node, 186–192
  - SwingCheckBox component, 173–176
  - SwingComboBox and SwingComboBoxItem components, 176–180
  - syntax, 23
- 
- T**
- target parameter, 272–273
  - text and Text node
    - adding, 31–38
    - bind with, 46–49
    - document files, 15
    - sayHello, 44
    - writing to screen, 28–29
  - text property
    - SwingButton, 168
    - SwingCheckBox, 174
  - TextBox
    - horizontal layout, 221–225
    - nested layouts, 227–230
    - vertical layout, 225–227
  - textFill property, 238
  - thickness of lines, 57
  - three-color LinearGradient, 79–81
  - timelines, 133–139
  - timers property, 266
  - timing for smooth animation, 133
  - titles in Stage, 30–31
  - toMillis method, 211
  - transformations
    - description, 125
    - PerspectiveTransform, 128–129
    - rotation, 127–128
    - XY, 125–127
  - transforms property, 256
  - translateX property
    - description, 256
    - Image View node, 127
  - translateY property
    - description, 256
    - frames, 137
    - Image View node, 127
  - transparency, 76–77

tween notation, 137  
tweening, 137  
two-color LinearGradient, 78–79  
.txt files, 15  
type names in nodes, 23  
<type> notation, 104  
types of variables, 46–47

## U

---

U-shaped drawing, 59–61  
underscores (\_), 35  
url parameter, 87, 89

## V

---

valueOf method, 212  
values in name-value pairs, 23  
var keyword, 46  
variables, 46–48  
VBox layout, 225–227  
-verbose parameter  
    javafx, 275  
    javafxc, 270  
-version parameter  
    javafx, 275–276  
    javafxc, 273  
versions in comments, 19  
vertical layout, 225–227  
video  
    overview, 201–202  
    play/pause button, 207–211  
    playing, 203–207  
    progress indicators, 211–215  
    supported files, 204  
visible property, 256  
VK\_ key codes, 258–265

void functions, 42  
volume property, 216, 266

## W

---

web color hex values, 76  
Web image files, 88  
Web Run configuration, 37  
wheelRotation event, 257  
whitespace in arrays, 60  
width  
    arcs, 67  
    DropShadow effect, 116  
    images, 88  
    lines, 57  
    rectangles, 62  
words, writing to screen, 28–29  
working files, 15  
writing to screen, 28–29

## X

---

x attribute, 62  
x event, 257  
-X parameter  
    javafx, 277  
    javafxc, 273  
x position in Cartesian coordinates, 33  
.xlsx files, 15  
XY transformations, 125–127

## Y

---

y attribute, 62  
y event, 257  
y position in Cartesian coordinates, 33