



Essential JavaFX™

Gail Anderson • Paul Anderson



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

Sun Microsystems, Inc. has intellectual property rights relating to implementations of the technology described in this publication. In particular, and without limitation, these intellectual property rights may include one or more U.S. patents, foreign patents, or pending applications.

Sun, Sun Microsystems, the Sun logo, J2ME, J2EE, Java Card, and all Sun and Java based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact: U.S. Corporate and Government Sales, (800) 382-3419, corpsales@pearsontechgroup.com.

For sales outside the United States please contact: International Sales, international@pearsoned.com.

A control number for this work is on file with The Library of Congress.

Copyright © 2009 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, California 95054 U.S.A.
All rights reserved.

Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to: Pearson Education, Inc., Rights and Contracts Department, 501 Boylston Street, Suite 900, Boston, MA 02116, Fax: (617) 671-3447.

ISBN-13: 978-0-13-704279-1

ISBN-10: 0-13-704279-5

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.
First printing, May 2009

Preface

As we complete the final edits and our printing deadline looms, we're excited and grateful to be involved with JavaFX. In February 2009, JavaFX reached the 100,000,000th download of the JavaFX runtime.¹ The ranks of JavaFX developers will undoubtedly grow as more developers see the flexibility and power of JavaFX. This synergy, we believe, will fuel continued development of the language and enhancements to the runtime environment.

This book is designed to get you up to speed quickly with JavaFX. JavaFX is a scripting language. It's not Java, but it's built on top of the Java runtime. You don't need experience with Java to succeed with JavaFX. Indeed, JavaFX's declarative syntax makes life easier if you *don't* think like a developer. Instead, JavaFX encourages you to think like a designer.

What does it mean to "think like a designer"? Basically, it means to visualize the structure of your application or widget and compose your scene out of simple shapes and other building blocks. In JavaFX, you compose a scene by declaring objects.

Let's take an example. Say you visualize a sky with the sun, the sea, and an island (think South Pacific). The sky is the background, reflecting the blues of a bright cloudless day (think of a linear gradient, going from "blue sky" to "azure"). The sun is a Circle, with a radial gradient consisting of yellows and oranges. The island is a quadratic curve (think of a cone-shaped volcano-type island paradise filled with a gradient of rich browns and tropical greens). And there you have your scene, as shown in Figure 1 (in a black and white approximation).²

Not only can you declare visual objects with JavaFX, but you can also declare animations. Animations give your objects life. Returning to our island paradise, visualize the beginning of the day. The colors are muted as the morning light slowly gives shape to an ethereal world. The sun rises and the island takes form. The sun continues

1. Jonathan Schwartz's Blog: *JavaFX Hits 100,000,000 Milestone!* February 13, 2009. URL: http://blogs.sun.com/jonathan/entry/javafx_hits_100_000_000

2. You'll find widget Island Paradise with the other JavaFX examples on the authors' web site at <http://www.asgteach.com/javafx>.

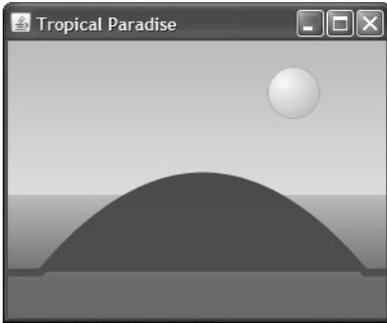


Figure 1. Visualizing an island paradise

higher in the sky and the whole world brightens. When the sun reaches its zenith, the colors of the sea and sky are saturated with “sea green,” “azure,” and “sky blue.” But, it’s a temporary brightness. The sun follows its path and eventually falls back into the sea in a glow of warm reds. The sky darkens. The sea fades. Eventually the island disappears. Everything is black. You build these behaviors with animation and a powerful JavaFX language construct called *binding*. (Binding lets you declare dependencies among objects—when a variable changes, JavaFX automatically updates all objects bound to that variable.)

JavaFX animation lets you move objects along a path (the sun rises and sets in an arc) and fade objects in and out with timelines. Our “island paradise” controls day and night with black and red rectangle “filters.” These filters color the scene as night, dawn, daytime, evening, dusk, and back to night, all cycling through an accelerated Circadian clock. Animation and binding make it all work.

If you’re a Java programmer, you will feel at home in JavaFX with packages and import statements, classes, things called public, and static type checking. (Don’t worry about these things if you’re not a Java programmer.) If you’re a JavaScript programmer, you will appreciate the value of static type checking coupled with a sophisticated type inference engine in JavaFX. (Type inference eases the burden of having to specify types everywhere.) But most importantly, we hope you’ll appreciate the simplicity of the JavaFX declarative style. For example, take this one-line JavaFX object “literal.”

```
Circle { centerX: 200 centerY: 40 radius: 25 fill: Color.YELLOW }
```

If you think the above describes a yellow circle, then you’re on your way! And, if you think perhaps it describes a yellow sun, that’s even better.

About the Audience

This book is aimed at developers with some previous programming experience (in any language). We don't assume you know Java and we assume you've never written a JavaFX script before. (The term *script* in this book refers to both the programs you write and the individual files that contain these programs.)

We hope to show you how to use JavaFX effectively. There is a diversity to JavaFX: you can use it to build games, create effective web-service-based widgets, or build snazzy front-ends to your desktop applications. You can use Swing-based components, “native” JavaFX components, or roll your own. You can collaborate with designers and import images and other assets to incorporate into your scene graph. Our aim is to expose some of this diversity so that you can forge ahead with your own successful JavaFX projects.

How to Use This Book

Chapter 1 gets you started with JavaFX. We show you how to download JavaFX and begin building projects with the NetBeans IDE. (We use NetBeans to build our examples, but you can also use Eclipse.)

Chapter 2 gives you a broad overview of JavaFX. It takes you through an example (a Guitar Tuner), pointing out how things are done with JavaFX. If you want to get a “feel” for the language, this chapter introduces you to many trademark JavaFX features.

Chapter 3 through Chapter 5 are “reference-oriented” chapters. Chapter 3 describes the JavaFX language, Chapter 4 describes graphical objects, and Chapter 5 discusses user interface components. These chapters are organized with small examples to help you find information quickly (how do I bind an object or generate a sequence with a for loop?). The language chapter covers everything from JavaFX built-in types to mixin inheritance. Graphical objects are the basic JavaFX shapes you use to build scene graphs and layout objects (islands in the sun, for example). The components chapter shows you the JavaFX Swing components and the JavaFX “native” UI components. We also show you how to build custom UI components in a more advanced section.

Chapter 6 shows you how to design and structure a JavaFX application. It introduces a building-block approach with a nod towards object oriented design principles.

Chapter 7 is all about JavaFX animation and timelines. JavaFX animation is both powerful and flexible. Transitions are “pre-packaged high-level” animations that help build straightforward motions quickly, such as fade-ins and fade-outs, scaling, and moving.

Chapter 8 discusses viewing and manipulating images. One example shows you how to design an animated photo carousel.

Chapter 9 covers web services. JavaFX provides two important utility classes that make it easier to work with web services. An `HttpRequest` class handles asynchronous web requests and a `PullParser` class simplifies processing the response data. We take you through several Flickr-based web service API calls.

Chapter 10 discusses the JavaFX mobile environment and explores the differences between desktop JavaFX and the JavaFX mobile runtime. We discuss guidelines for targeting mobile devices and how to make an application mobile-friendly.

About the Examples

You can download the source code for all book examples from the authors' web site at

<http://www.asgteach.com/javafx>

In addition, example applications are deployed so you can try them out.

Notational Conventions

We've applied a rather light hand with font conventions in an attempt to keep the page uncluttered. Here are the conventions we follow.

Element	Font Example
JavaFX class	Shape, Circle, Color
JavaFX property	layoutBounds, opacity, height
JavaFX code	<pre>def sunPath = Path { elements: sunElements stroke: Color.GRAY }</pre>
URL	http://javafx.com/
file name	Main.fx , Carousel.fx
key combinations	Ctrl+Space
NetBeans menu selections	Properties menu item
code within text	The animation varies property <code>opacity</code> from . . .
code highlighting (to show modified or relevant portions)	<pre>def sunPath = Path { elements: sunElements stroke: Color.GRAY }</pre>

2 A Taste of JavaFX

As the preface hints, JavaFX has a combination of features that makes it unique. This chapter gives you a taste of the language and some of these features. Our goal is to choose a representative example so you get a feel for the kinds of programs possible with JavaFX. The example (a guitar tuner) illustrates language constructs while keeping the discussion concrete. We'll veer away from the example at times to illustrate additional JavaFX features that are relevant. While this overview is in no way complete (remember, it's just a taste), we hope to entice you to explore JavaFX further.

The source code for `GuitarTuner` appears at the end of the chapter (see “Source Code for Project `GuitarTuner`” on page 36). To keep the text flowing, we'll show snippets from this application throughout the overview.

What You Will Learn

- What makes JavaFX unique as a scripting language
- All about object literals and declarative constructs
- Introducing the JavaFX scene graph
- Declaring variables, properties, and objects
- Initializing objects and object properties
- Basics in container coordinate space and layout
- Creating a custom node
- Manipulating objects with color, effects, and gradients
- Getting things done with binding, event handlers, and animation

2.1 Introducing JavaFX

What is JavaFX? JavaFX is a scripting language with static typing. You can call a Java API as needed from JavaFX and create new object types with classes, but JavaFX also provides an easy declarative syntax. (Declarative means you say what you want and

the system figures out how to do it for you.) JavaFX provides properties for manipulating objects within a 2D coordinate system, specifying fill and pen stroke colors, and creating special effects. You can create shapes and lines, manipulate images, play videos and sounds, and define animations.

Let's begin exploring JavaFX by introducing the basics. Our introduction begins with project GuitarTuner where you'll see the main structure of a JavaFX program. Then, you'll explore a few JavaFX language constructs and see how to improve the appearance of your applications. Finally, you'll see how to make applications do things.

JavaFX in a Nutshell

JavaFX is statically typed, meaning program data types are known at compile time. JavaFX also uses type inference. This means you don't have to declare the type of every variable because JavaFX can generally figure it out for you. This gives JavaFX the efficiency of a statically typed language combined with the ease of a declarative language.

2.2 Project GuitarTuner

Project GuitarTuner helps you tune your guitar. It displays a visual guitar fret board with six strings. The letter (note) corresponding to the guitar string appears next to the fret board. When you click a string with the mouse, you'll hear a synthesized guitar note for the selected string as it vibrates visually. Project GuitarTuner uses the Java `javax.sound.midi` API to generate the sounds. Figure 2.1 shows this application running when the A string is vibrating. The corresponding JavaFX graphical objects are labeled.

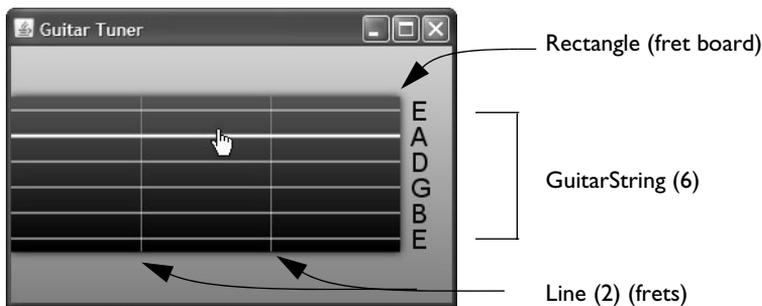


Figure 2.1 JavaFX application GuitarTuner

The Scene Graph Metaphor

JavaFX programs with a graphical user interface define a *stage* and a *scene* within that stage. The stage represents the top level container for all JavaFX objects; that is, the content area for an applet or the frame for a widget. The central metaphor in JavaFX for specifying graphics and user interaction is a *scene graph*. A scene defines a hierarchical node structure that contains all the scene's components. *Nodes* are represented by graphical objects, such as geometric shapes (Circle, Rectangle), text, UI controls, image viewers, video viewers, and user-created objects (such as GuitarString in our example). Nodes can also be containers that in turn hold more nodes, letting you group nodes together in hierarchical structures. (For example, Group is a general-purpose container node, HBox provides horizontal layout alignment, and VBox provides vertical layout alignment.) The scene graph is this hierarchical node structure.

Figure 2.2 shows the scene graph for project GuitarTuner. Compare the visual graphical elements in Figure 2.1 with the scene graph depicted in Figure 2.2.

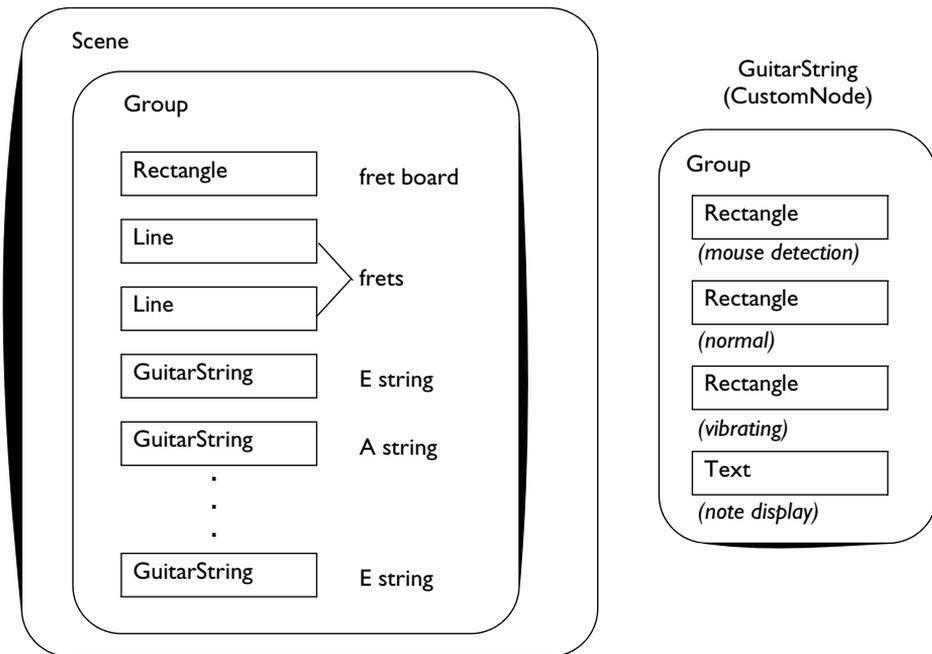


Figure 2.2 Nested Scene Graph for GuitarTuner

In general, to construct a JavaFX application, you build the scene graph, specifying the look and behavior of all its nodes. Then, your application just “runs.” Some applications need input to go—user actions that activate animations or affect component

properties. Other applications just run on their own. (Building the scene graph is analogous to winding up a toy. When you're done, the application just runs.)

JavaFX Scene Graph

The power of the scene graph is that, not only do you capture the entire structure of your application in a data structure, but you can change the display simply by modifying properties of the objects in the scene graph. (For example, if you change a node's `visible` property to `false`, that node, and any nodes it contains, disappears. If you change a node's location, it moves.)

Within the scene graph for project `GuitarTuner`, you see the `Scene` at the top level, which contains a `Group`. Within the `Group` there is a `Rectangle` for the fret board (the guitar neck), two `Line` nodes representing frets, and six `GuitarStrings`. Each `GuitarString` is in turn its own `Group` consisting of three `Rectangles` and a `Text` node. Nodes that contain other nodes (such as `Scene` and `Group`) include a `content` property that holds subnodes. The hierarchical nature of the scene graph means that all nodes at the same level share the same coordinate space. You therefore build node structures (such as `GuitarString`) that use a relative coordinate system. You'll see shortly why this is useful.

Think Like A Designer

JavaFX encourages you to think like a designer. As a first step, visualize the structure of your application or widget and compose your scene out of simple shapes and other building blocks.

The order of nodes within a parent container affects their rendering. That is, the first node in the container is "drawn" first. The final node is "drawn" last and is on top of the view. Nodes (depending on their placement within the coordinate system) may visually block or "clip" previously drawn nodes. In `GuitarTuner`, the nodes must be in a specific order. You draw the fret board first, then the frets, and finally the guitar strings, which appear on top.

Changing the relative order of nodes in a container is easy. The `toFront()` function brings a node to the front (top) and the `toBack()` function sends a node to the back (bottom).

Hierarchical Scene Graph

Figure 2.3 also shows a scene graph of project `GuitarTuner`. Figure 2.2 and Figure 2.3 depict the same structure, but Figure 2.3 shows the hierarchical relationship among the nodes in the scene using a graphical tree view. Nodes at the same level share the same coordinate space. For example, the three `Rectangles` and `Text` nodes in the `GuitarString` share the same coordinate system.

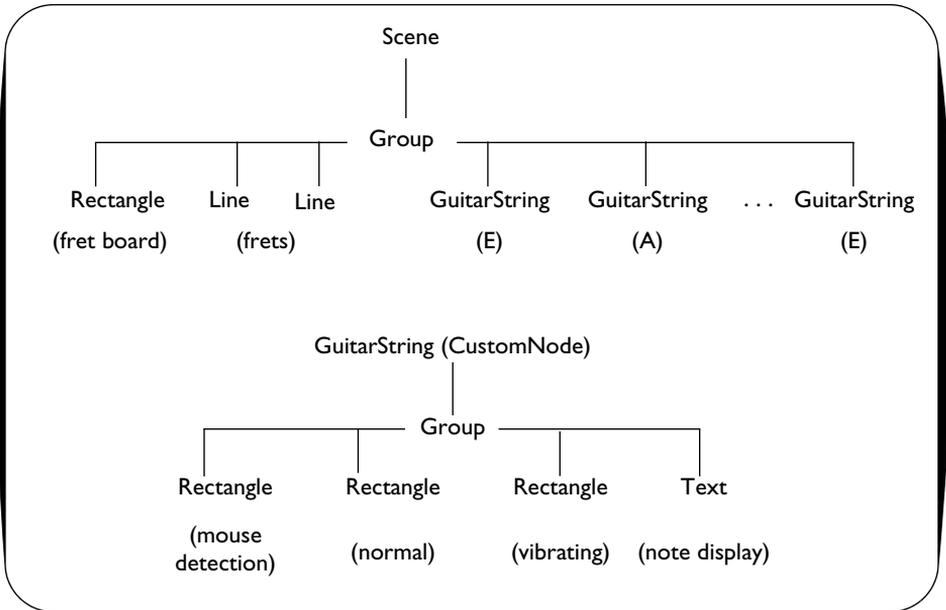


Figure 2.3 Scene Node Graph for project GuitarTuner

2.3 JavaFX Program Structure

JavaFX program structure is simple. For programmers who are used to traditionally compiled programs, programming in JavaFX will feel different. With static typing, JavaFX gives you feedback at compile time when you use types incorrectly. This greatly enhances your ability to write correct code. Furthermore, with the NetBeans IDE, you can access JavaDocs for all JavaFX types (classes) and dynamically query these class properties and functions, essentially getting feedback at edit time.

Let's see how the Stage and Scene form the JavaFX program structure.

Stage and Scene

The Stage is the top-level container and contains the Scene. The Scene, in turn, holds nodes that make up the scene graph. Every JavaFX program that has graphical objects declares a Stage object.

Here is a top-level implementation of the scene graph for GuitarTuner from Figure 2.2 (or Figure 2.3). (We'll look at GuitarString's node graph shortly.)

```

// Stage and Scene Graph
Stage {
  title: "Guitar Tuner"
  Scene {
    // content is sequence of SceneGraph nodes
    content: [
      Group {
        content: [
          Rectangle { ... }
          Line { ... }
          Line { ... }
          GuitarString { ... }
        ]
      } // Group
    ]
  } // Scene
} // Stage

```

Object Literals

The Stage and Scene objects are instantiated with object literal expressions, or *object literals*. Object literals provide a declarative style of programming. Intuitively, declarative means “tell me what you want, not how to do it.” As you will see, the real declarative part of JavaFX is *binding*. We show why this is so powerful later in the chapter.

Object literals require an object (or class) type (such as Stage or Scene) followed by curly braces { }. Any properties you need to initialize appear inside the braces. (Stage has a title property and Scene and Group both have content properties.) Each property has a name, followed by a colon : and an initial value for the property. You separate properties with commas, line breaks, or white space. Here, for example is an object literal that initializes a Rectangle (properties x and y designate the upper-left corner origin).

```
Rectangle { x: 10, y: 20, height: 15, width: 150 }
```

The above Stage, Scene, and Group objects are defined with object literals. Note that the Scene object nests inside the Stage object. Likewise, a Group nests inside the Scene. Square brackets [] define a sequence of items for the content property in a Scene or Group. Here, the Scene object’s content property is a sequence of all of the top-level nodes of the Scene. In the GuitarTuner application, this is a Group node (see Figure 2.2 or Figure 2.3). The Group node likewise includes a content property with all of its subnodes (Rectangles, Lines, and a custom GuitarString). How you nest these nodes determines the structure of the scene graph.

Here's the top-level implementation for GuitarString from its scene graph in Figure 2.2 (and Figure 2.3).

```
// GuitarString - defined as custom class
Group {
    content: [
        Rectangle { ... }
        Rectangle { ... }
        Rectangle { ... }
        Text { ... }
    ]
} // Group
```

The GuitarString consists of a Group node whose content property defines a sequence containing three rectangles and a Text object. You'll see how this fits into the Guitar-Tuner application later on.

2.4 Key JavaFX Features

GuitarTuner is a fairly typical JavaFX example application. It has a graphical representation and responds to user input by changing some of its visual properties (as well as producing guitar sounds). Let's look at some of the key JavaFX features it uses to give you a broad look at the language.

Signature JavaFX Features

Included in any list of key JavaFX features are binding, node event handlers, and animation. We discuss each of these important constructs in their own section (see "Doing Things" on page 31).

Type Inference

JavaFX provides `def` for read-only variables and `var` for modifiable variables.

```
def numberFrets = 2;           // read-only Integer
var x = 27.5;                  // variable Number
var y: Number;                 // default value is 0.0
var s: String;                 // default value is ""
```

The compiler *infers types* from the values you assign to variables. Read-only `numberFrets` has inferred type `Integer`; variable `x` has inferred type `Number` (`Float`). This means you don't have to specify types everywhere (and the compiler tells you when a type is required.)

Strings

JavaFX supports dynamic string building. Curly braces { } within a String expression evaluate to the contents of the enclosed variable. You can build Strings by concatenating these String expressions and String literals. For example, the following snippet prints "Greetings, John Doe!".

```
def s1 = "John Doe";  
println("Greetings, {s1}!");    // Greetings, John Doe!
```

Shapes

JavaFX has numerous shapes that help you create scene graph nodes. There are shapes for creating lines (*Line*, *CubicCurve*, *QuadCurve*, *PolyLine*, *Path*) and shapes for creating geometric figures (*Arc*, *Circle*, *Ellipse*, *Rectangle*, *Polygon*). The Guitar-Tuner application uses only *Rectangle* and *Line*, but you'll see other shape examples throughout this book.

Let's look at shapes *Rectangle* and *Circle*. They are both standard JavaFX shapes that extend class *Shape* (in package `javafx.scene.shape`). You define a *Circle* by specifying values for its `radius`, `centerX`, and `centerY` properties. With *Rectangle*, you specify values for properties `height`, `width`, `x`, and `y`.

Shapes share several properties in common, including properties `fill` (type *Paint* to fill the interior of the shape), `stroke` (type *Paint* to provide the outline of the shape), and `strokeWidth` (an *Integer* for the width of the outline).

Here, for example, is a *Circle* with its center at point (50, 50), radius 30, and color `Color.RED`.

```
Circle {  
    radius: 30  
    centerX: 50  
    centerY: 50  
    fill: Color.RED  
}
```

Here is a *Rectangle* with its top left corner at point (30, 100), height 30, width 80, and color `Color.BLUE`.

```
Rectangle {  
    x: 30, y: 100  
    height: 30, width: 80  
    fill: Color.BLUE  
}
```

All shapes are also *Nodes* (`javafx.scene.Node`). *Node* is an all-important class that provides local geometry for node elements, properties to specify transformations

(such as translation, rotation, scaling, or shearing), and properties to specify functions for mouse and key events. Nodes also have properties that let you assign CSS styles to specify rendering.¹ We discuss graphical objects in detail in Chapter 4.

Sequences

Sequences let you define a collection of objects that you can access sequentially. You must declare the type of object a sequence will hold or provide values so that its type can be inferred. For example, the following statements define sequence variables of `GuitarString` and `Rectangle` objects.

```
var guitarStrings: GuitarString[];
var rectangleSequence: Rectangle[];
```

These statements create read-only sequences with `def`. Here, sequence `noteValues` has an inferred type of `Integer[]`; sequence `guitarNotes` has an inferred type of `String[]`.

```
def noteValues = [ 40,45,50,55,59,64 ];
def guitarNotes = [ "E", "A", "D", "G", "B", "E" ];
```

Sequences have specialized operators and syntax. You will use sequences in JavaFX whenever you need to keep track of multiple items of the same object type. The `GuitarTuner` application uses a sequence with a `for` loop to build multiple `Line` objects (the frets) and `GuitarString` objects.

```
// Build Frets
for (i in [0..numberFrets])
    Line { . . . }

// Build Strings
for (i in [0..numberStrings])
    GuitarString { . . . }
```

The notation `[0..n]` is a sequence literal and defines a range of numbers from 0 to `n-1`, inclusive.

You can declare and populate sequences easily. The following *declarative approach* inserts `Rectangles` into a sequence called `rectangleSequence`, stacking six `Rectangles` vertically.

```
def rectangleSequence = for (i in [0..5])
    Rectangle {
```

1. Cascading Style Sheets (CSS) help style web pages and let designers give a uniform look and feel throughout an application, widget, or entire web site. You can use CSS to similarly style JavaFX nodes. (See “Cascading Style Sheets (CSS)” on page 148 for details on applying styles to JavaFX nodes.)

```

    x: 20
    y: i * 30
    height: 20
    width: 40
}

```

You can also insert number values or objects into an existing sequence using the insert operator. The following *imperative approach* inserts the six Rectangles into a sequence called `varRectangleSequence`.

```

var varRectangleSequence: Rectangle[];
for (i in [0..5])
    insert Rectangle {
        x: 20
        y: i * 30
        height: 20
        width: 40
    } into varRectangleSequence;

```

JavaFX Tip

The declarative approach with `rectangleSequence` is always preferred (if possible). By using `def` rather than `var` and declaring sequences rather than inserting objects into them, type inference will more likely help you and the compiler can optimize the code more effectively.

You'll see more uses of sequence types throughout this book.

Calling Java APIs

You can call any Java API method in JavaFX programs without having to do anything special. The `GuitarString` node “plays a note” by calling function `noteOn` found in Java class `SingleNote`. Here is `GuitarString` function `playNote` which invokes `SingleNote` member function `noteOn`.

```

function playNote(): Void {
    synthNote.noteOn(note); // nothing special to call Java methods
    vibrateOn();
}

```

Class `SingleNote` uses the Java `javax.sound.midi` package to generate a synthesized note with a certain value (60 is “middle C”). Java class `SingleNote` is part of project `GuitarTuner`.

Extending CustomNode

JavaFX offers developers such object-oriented features as user-defined classes, overriding virtual functions, and abstract base classes (there is also “mixin” inheritance).

GuitarTuner uses a class hierarchy with subclass `GuitarString` inheriting from a JavaFX class called `CustomNode`, as shown in Figure 2.4.

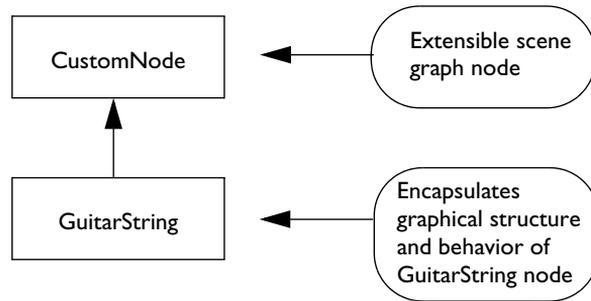


Figure 2.4 `GuitarString` Class Hierarchy

This approach lets you build your own graphical objects. In order for a custom object to fit seamlessly into a JavaFX scene graph, you base its behavior on a special class provided by JavaFX, `CustomNode`. Class `CustomNode` is a scene graph node (a type of `Node`, discussed earlier) that lets you specify new classes that *extend* from it. Just like Java, “extends” is the JavaFX language construct that creates an inheritance relationship. Here, `GuitarString` extends (*inherits from*) `CustomNode`. You then supply the additional structure and behavior you need for `GuitarString` objects and override any functions required by `CustomNode`. JavaFX class constructs are discussed in more detail in Chapter 3 (see “Classes and Objects” on page 67).

Here is some of the code from `GuitarTuner`'s `GuitarString` class. The `create` function returns a `Node` defining the `Group` scene graph for `GuitarString`. (This scene graph matches the node structure in Figure 2.2 on page 15 and Figure 2.3 on page 17. Listing 2.2 on page 38 shows the `create` function in more detail.)

```

public class GuitarString extends CustomNode {
    // properties, variables, functions
    . . .
    protected override function create(): Node {
        return Group {
            content: [
                Rectangle { ... }
                Rectangle { ... }
                Rectangle { ... }
                Text { ... }
            ]
        } // Group
    }
} // GuitarString
  
```

Geometry System

In JavaFX, nodes are positioned on a two-dimensional coordinate system with the origin at the upper-left corner. Values for x increase horizontally from left to right and y values increase vertically from top to bottom. The coordinate system is always relative to the parent container.

Layout/Groups

Layout components specify how you want objects drawn relative to other objects. For example, layout component `HBox` (horizontal box) evenly spaces its subnodes in a single row. Layout component `VBox` (vertical box) evenly spaces its subnodes in a single column. Other layout choices are `Flow`, `Tile`, and `Stack` (see “Layout Components” on page 119). You can nest layout components as needed.

Grouping nodes into a single entity makes it straightforward to control event handling, animation, group-level properties, and layout for the group as a whole. Each group (or layout node) defines a coordinate system that is used by all of its children. In `GuitarTuner`, the top level node in the scene graph is a `Group` which is centered vertically within the scene. The subnodes are all drawn relative to the origin $(0, 0)$ within the top-level `Group`. Centering the `Group`, therefore, centers its contents as a whole.

Benefits of Relative Coordinate Space

Nodes with the same parent share the same relative coordinate space. This keeps any coordinate space calculations for subnodes separate from layout issues of the parent container. Then, when you move the parent, everything under it moves, keeping relative positions intact.

JavaFX Script Artifacts

Defining the `Stage` and `Scene` are central to most JavaFX applications. However, JavaFX scripts can also contain package declarations, import statements, class declarations, functions, variable declarations, statements, and object literal expressions. You’ve already seen how object literal expressions can initialize nodes in a scene graph. Let’s discuss briefly how you can use these other artifacts.

Since JavaFX is statically typed, you must use either import statements or declare all types that are not built-in. You’ll typically define a package and then specify import statements. (We discuss working with packages in Chapter 3. See “Script Files and Packages” on page 86.) Here is the package declaration and import statements for `GuitarTuner`.

```
package guitartuner;  
  
import javafx.scene.effect.DropShadow;
```

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

    . . . more import statements . . .

import javafx.stage.Stage;
import noteplayer.SingleNote;
```

If you're using NetBeans, the IDE can generate import statements for you (type **Ctrl+Shift+I** in the editor window).

You'll need script-level variables to store data and read-only variables (`def`) for values that don't change. In `GuitarTuner`, we define several read-only variables that help build the guitar strings and a variable (`singleNote`) that communicates with the Java midi API. Note that `noteValues` and `guitarNotes` are `def` sequence types.

```
def noteValues = [ 40,45,50,55,59,64 ];
def guitarNotes = [ "E","A","D","G","B","E" ];
def numberFrets = 2;
def numberStrings = 6;
var singleNote = SingleNote { };
```

When you declare a `Stage`, you define the nested nodes in the scene graph. Instead of declaring nodes only as object literal expressions, it's also possible to assign these object literals to variables. This lets you refer to them later in your code. (For example, the `Scene` object literal and the `Group` object literal are assigned to variables in order to compute the offset for centering the group vertically in the scene.)

```
var scene: Scene;
var group: Group;

scene: scene = Scene { ... }
group = Group { ... }
```

You may also need to execute JavaFX script statements or define utility functions. Here's how `GuitarTuner` makes the `SingleNote` object emit a "guitar" sound.

```
singleNote.setInstrument(27);           // "Clean Guitar"
```

Once you set up the `Stage` and scene graph for an application, it's ready to ready to run.² In `GuitarTuner`, the application waits for the user to pluck (click) a guitar string.

2. Java developers may wonder where function `main()` is. As it turns out, the JavaFX compiler generates a `main()` for you, but from a developer's view, you have just a script file.

2.5 Making Things Look Good

Using JavaFX features that enhance the appearance of graphical objects will help your application look professionally designed. Here are some simple additions you can apply.

Gradients

Gradients lend a depth to surfaces and backgrounds by gradually varying the color of the object's `fill` property. In general, use linear gradients with rectangular shapes and radial gradients with circles and ovals. In `GuitarTuner`, the background is a linear gradient that transitions from `Color.LIGHTGRAY` (at the top) to the darker `Color.GRAY` (at the bottom) as shown in Figure 2.5. The guitar fret board also uses a linear gradient.

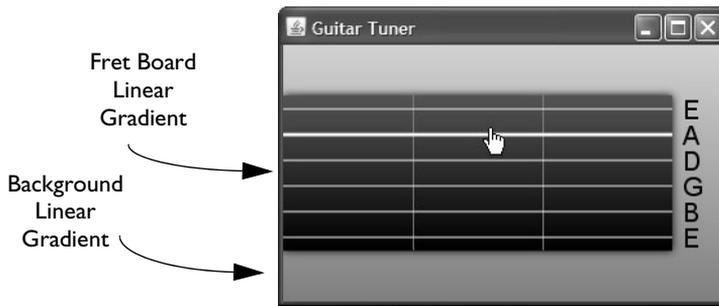


Figure 2.5 Gradients in the `GuitarTuner` Application

Here is the `LinearGradient` for the background scene in `GuitarTuner`, defined for property `fill`. Note that specifying gradients is declarative; you identify the look you want and the system figures out how to achieve it, independent of screen resolution, color depth, etc.

```
fill: LinearGradient {
    startX: 0.0
    startY: 0.0
    endX: 0.0
    endY: 1.0
    proportional: true
    stops: [
        Stop {
            offset: 0.0
            color: Color.LIGHTGRAY
        },
        Stop {
            offset: 1.0
```

```

        color: Color.GRAY
    }
]
}

```

The background gradient changes color along the y axis and the color is constant along the x axis (properties `startX` and `endX` are the same). Property `stops` is a sequence of `Stop` objects containing an `offset` and a `color`. The `offset` is a value between 0 and 1 inclusive; each succeeding `offset` must have a higher value than the preceding one.

Property `proportional` indicates whether `start` and `end` values are proportional (defined between `[0..1]` if `true`) or absolute (absolute coordinates if `false`).

Radial gradients work well for circular shapes, as shown in Figure 2.6. Here you see three `Circle` shapes, all with radial gradients. The first circle defines a gradient with its center in the lower left quadrant (`centerX` is 0.25 and `centerY` is 0.75). The second circle's gradient is centered (`centerX` and `centerY` are both 0.5), and the third circle's gradient appears in the upper right quadrant (`centerX` is 0.75 and `centerY` is 0.25).

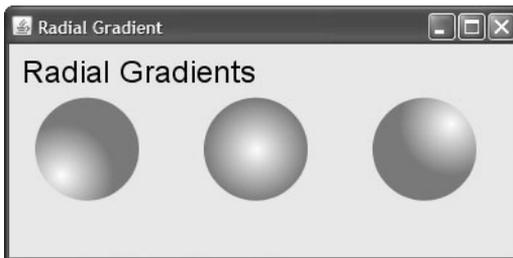


Figure 2.6 Radial Gradients work well with circular shapes

Here is the radial gradient for the middle circle.

```

fill: RadialGradient {
    centerX: 0.5    // x center of gradient
    centerY: 0.5   // y center of gradient
    radius: 0.5    // radius of gradient
    stops: [
        Stop {
            offset: 0
            color: Color.WHITE
        },
        Stop {
            offset: 1

```

```

        color: Color.DODGERBLUE
    }
}
}

```

Note that the gradient is half the size of the circle (radius is 0.5). Making the gradient less than the full size lets the last stop color appear more prominent (the dark color predominates).

Color

You specify a shape’s color with property `fill`. JavaFX has many predefined colors ranging alphabetically from `Color.ALICEBLUE` to `Color.YELLOWGREEN`. (In the NetBeans IDE, press **Ctrl+Space** when the cursor is after the dot in `Color` to see a complete list, as shown in Figure 2.7.)

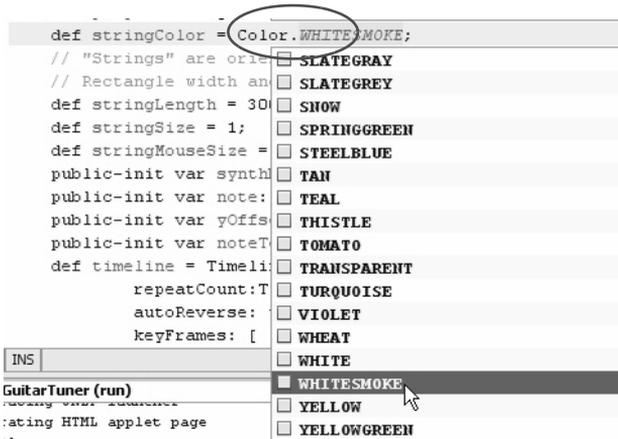


Figure 2.7 Explore color choices with the NetBeans IDE

You can also specify arbitrary colors with `Color.rgb` (each RGB value ranges from 0 to 255), `Color.color` (each RGB value ranges from 0 to 1), and `Color.web` (a String corresponding to the traditional hexadecimal-based triad). An optional final argument sets the opacity, where 1 is fully opaque and 0 is fully translucent. You can also make a shape transparent by setting its `fill` property to `Color.TRANSPARENT`.

Here are several examples of color settings. Each example sets the opacity to .5, which allows some of the background color to show through.

```

def c1 = Color.rgb(10, 255, 15, .5); // bright lime green
def c2 = Color.color(0.5, 0.1, 0.1, .5); // dark red
def c3 = Color.web("#546270", .5); // dark blue-gray

```

Numeric-based color values (rather than hexadecimal strings or predefined colors) let you write functions and animations that numerically manipulate gradients, colors, or opacity. For example, the following `fill` property gets its `Color.rgb` values from a `for` loop's changing value `i`. The loop produces three different shades of green, depending on the value of `i`.

```
def rectangleSequence = for (i in [0..2])
  Rectangle {
    x: 60 * i
    y: 50
    height: 50
    width: 40
    fill: Color.rgb(10 + (i*50), 100 + (i*40), i*50)
  }
```

Figure 2.8 shows the resulting set of rectangles with different `fill` values.

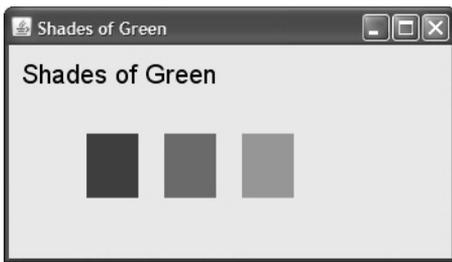


Figure 2.8 Manipulating numeric-based Color values

Rectangles with Arcs

You can soften the corners of Rectangles by specifying properties `arcWidth` and `arcHeight`, as shown in Figure 2.9. The first Rectangle has regular, square corners. The second Rectangle sets `arcHeight` and `arcWidth` to 15, and the third one uses value 30 for both. Here's the object literal for the third Rectangle.

```
Rectangle {
  x: 180
  y: 0
  height: 70
  width: 60
  arcHeight: 30
  arcWidth: 30
  fill: LinearGradient { . . . }
}
```

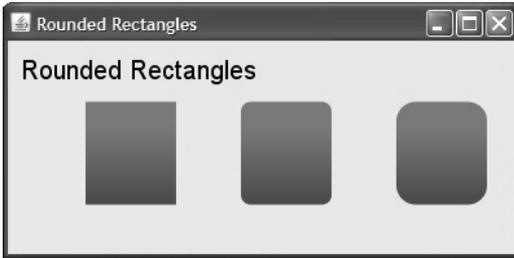


Figure 2.9 Soften Rectangles with rounded corners

DropShadows

One of the many effects you can specify is `DropShadow` (effects are declarative). Effect `DropShadow` applies a shadow to its node, giving the node a three-dimensional look. In project `GuitarTuner`, the fret board (guitar neck) uses a default drop shadow, as follows.

```
effect: DropShadow { }
```

The default object literal provides a drop shadow with these values.

```
effect: DropShadow {
    offsetX: 0.0
    offsetY: 0.0
    radius: 10.0
    color: Color.BLACK
    spread: 0.0
}
```

You can manipulate the location of the shadow by changing `offsetX` and `offsetY`. Negative values for `offsetY` set the shadow above the object and negative values for `offsetX` set the shadow to the left. Positive values for `offsetX` and `offsetY` place the shadow to the right and below, respectively. You can also change a shadow's size (radius), color, and spread (how "sharp" the shadow appears). A spread value of 1 means the shadow is sharply delineated. A value of 0 provides a "fuzzy" appearance. Figure 2.10 shows three rectangles with drop shadows that fall below and to the right of the rectangles, using these offsets.

```
effect: DropShadow {
    // shadow appears below and to the right of object
    offsetX: 5.0
    offsetY: 5.0
}
```



Figure 2.10 Drop shadows provide a three-dimensional effect

2.6 Doing Things

JavaFX has three main constructs for doing things: binding, node properties that define event handlers, and animation. Together, these constructs provide powerful yet elegant solutions for modifying scene graphs based on user input or other events. Let's see how GuitarTuner uses these constructs to get its tasks done.

Binding

Binding in JavaFX is a powerful technique and a concise alternative to specifying traditional callback event handlers. Basically, binding lets you make a property or variable depend on the value of an expression. When you update any of the “bound to” objects in the expression, the dependent object automatically changes. Suppose, for example, we bind area to height and width, as follows.

```
var height = 3.0;
var width = 4.0;
def area = bind height * width;           // area = 12.0

width = 2.5;                             // area = 7.5
height = 4;                               // area = 10.0
```

When either height or width changes, so does area. Once you bind a property (or variable), you can't update it directly. For example, you get a compile-time error if you try to directly update area.

```
area = 5;                                 // compile time error
```

If you make area a var and provide a binding expression, you'll get a runtime error if you try to update it directly.

In `GuitarTuner`, the vibrating string changes both its location (property `translateY`) and its thickness (property `height`) at run time to give the appearance of vibration. These properties are bound to other values that control how a guitar string node changes.

```
var vibrateH: Number;
var vibrateY: Number;

Rectangle {
    x: 0.0
    y: yOffset
    width: stringLength
    height: bind vibrateH // change height when vibrateH changes
    fill: stringColor
    visible: false
    translateY: bind vibrateY // change translateY when vibrateY changes
}
```

`GuitarTuner` also uses `bind` to keep the fret board centered vertically by binding property `layoutY` in the top level group.

```
group = Group {
    layoutY: bind (scene.height - group.layoutBounds.height) /
                2 - group.layoutBounds.minY
    . . .
}
```

Node property `layoutBounds` provides bounds information for its contents. If a user resizes the window, the top level group is automatically centered vertically on the screen. Binding helps reduce event processing code because (here, for example) you don't have to write an event handler to detect a change in the window size.

Binding is Good

Binding is good for many things. For example, you can change the appearance of a node based on changes to the program's state. You can make a component visible or hidden. You can also use binding to declaratively specify layout constraints. Not only does binding produce less code, but the code is less error-prone, easier to maintain, and often easier for the compiler to optimize.

Mouse Events

JavaFX nodes have properties for handling mouse and key events. These properties are set to callback functions that the system invokes when an event triggers. In `GuitarTuner`, the "mouse detection" rectangle has the following event handler to detect a mouse click event.

```

onMouseClicked: function(evt: MouseEvent): Void {
    if (evt.button == MouseButton.PRIMARY) {
        // play and vibrate selected "string"
    }
}

```

The if statement checks for a click of the primary mouse button (generally the left mouse button is primary) before processing the event. The event handler function (shown in the next section) plays the note and vibrates the string.

Animations

JavaFX specializes in animations. (In fact, we dedicate an entire chapter to animation. See Chapter 7 beginning on page 205.) You define animations with timelines and then invoke Timeline functions `play` or `playFromStart` (there are also functions `pause` and `stop`). Timelines consist of a sequence of key frame objects that define a frame at a specific time offset within the timeline. (Key frames are declarative. You say “this is the state of the scene at this key time” and let the system figure out how to render the affected objects.) Within each key frame, you specify values, an action, or both. Traditionally, people think of animations as a way to move objects. While this is true, you’ll see that JavaFX lets you animate any writable object property. You could, for instance, use animation to fade, rotate, resize, or even brighten an image.

Figure 2.11 shows a snapshot of a program with simple animation. It moves a circle back and forth across its container.

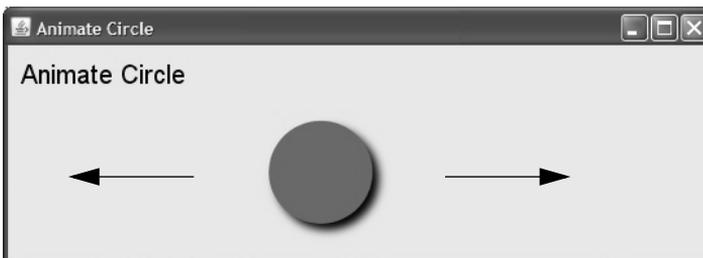


Figure 2.11 Timelines let you specify animations

Here is the timeline that implements this animation using a specialized shorthand notation for KeyFrames. The timeline starts out by setting variable `x` to 0. In gradual, linear increments, it changes `x` so that at four seconds, its value is 350. Now, it performs the action in reverse, gradually changing `x` so that in four more seconds it is back to 0 (autoReverse is true). This action is repeated indefinitely (or until the timeline is stopped or paused). Constants `0s` and `4s` are Duration literals.

```

var x: Number;
Timeline {
    repeatCount: Timeline.INDEFINITE
    autoReverse: true
    keyFrames: [
        at (0s) { x => 0.0 }
        at (4s) { x => 350 tween Interpolator.LINEAR }
    ]
}.play();           // start Timeline
. . .
Circle {
    . . .
    translateX: bind x
}

```

The JavaFX keyword `tween` is a key frame operator that lets you specify how a variable changes. Here, we use `Interpolator.LINEAR` for a linear change. That is, `x` doesn't jump from 0 to 350, but gradually takes on values in a linear fashion. Linear interpolation moves the `Circle` smoothly from 0 to 350, taking four seconds to complete one iteration of the timeline.

JavaFX has other interpolators. Interpolator `DISCRETE` jumps from the value of one key frame to the second. Interpolator `EASEIN` is similar to `LINEAR`, except the rate of change is slower at the onset. Similarly, `EASEOUT` is slower at the finish and `EASEBOTH` provides easing on both ends of the timeline.

To make this animation apply to the `Circle` node, you bind the `Circle`'s `translateX` property to the variable manipulated by the timeline (`x`). Property `translateX` represents a node's change in the `x` direction.

Now let's examine how `GuitarTuner` uses animation to vibrate the guitar string and play its note. Each `GuitarString` object uses two rectangles to implement its visible behavior. One rectangle is a stationary, thin "string" and represents the string in a static state. This motionless rectangle is always visible in the scene. The second rectangle is only visible when the string is "played." This rectangle expands and contracts its height quickly using animation (a `Timeline`). This moving rectangle gives users the illusion of a vibrating string.

To get a uniform vibrating effect, the rectangle must expand and contract evenly on the top and bottom. The animation makes the string appear to vibrate by varying the height of the rectangle from 1 to 3 while keeping it vertically centered by varying its `translateY` property between 5 and 4. When the string is clicked, the string's note plays and the rectangle vibrates for the allotted time. When the timeline stops, only the stationary rectangle is visible.

Let's first look at the timeline that plays the note. This timeline appears in the event handler for the GuitarString node (see the code for GuitarString in Listing 2.2 on page 38).

```
onMouseClicked: function(evt: MouseEvent): Void {
  if (evt.button == MouseButton.PRIMARY) {
    Timeline {
      keyFrames: [
        KeyFrame {
          time: 0s
          action: playNote // play note and start vibration
        }
        KeyFrame {
          time: 2.8s
          action: stopNote // stop playing note and stop vibration
        }
      ]
    }.play(); // start Timeline
  }
}
```

Here, the timeline is an object literal defined inside the event handler, invoked with function `play`. This timeline defines a sequence of `KeyFrame` objects, where function `playNote` is invoked at time offset 0 seconds and function `stopNote` is invoked at time offset 2.8 seconds (2.8s). Here are functions `playNote` and `stopNote`.

```
// play note and start vibration
function playNote(): Void {
  synthNote.noteOn(note);
  vibrateOn();
}

// stop playing note and stop vibration
function stopNote(): Void {
  synthNote.noteOff(note);
  vibrateOff();
}
```

Function `synthNote.noteOn` calls a Java class API to play the guitar string. Function `vibrateOn` causes the string vibration.

```
function vibrateOn(): Void {
  play.visible = true; // make the vibrating rectangle visible
  timeline.play(); // start the vibration timeline
}
```

Here is the vibration timeline.

```
def timeline = Timeline {
  repeatCount: Timeline.INDEFINITE
  autoReverse: true
  keyFrames: [
```

```

        at (0s) { vibrateH => 1.0 }
        at (.01s) { vibrateH => 3.0 tween Interpolator.LINEAR }
        at (0s) { vibrateY => 5.0 }
        at (.01s) { vibrateY => 4.0 tween Interpolator.LINEAR }
    ]
};

```

This timeline uses the shorthand notation discussed earlier for key frames and animates two variables: `vibrateH` and `vibrateY`. Variable `vibrateH` changes the height of the rectangle that represents the vibrating string. Variable `vibrateY` changes the vertical position of the rectangle to keep it centered as the oscillating height changes.

2.7 Source Code for Project GuitarTuner

Listing 2.1 and Listing 2.2 show the code for class `GuitarString` in two parts. Listing 2.1 includes the class declarations, functions, class-level variables, and properties for class `GuitarString`. Note that several variables are declared `public-init`. This JavaFX keyword means that users of the class can provide initial values with object literals, but otherwise these properties are read-only. The default accessibility for all variables is *script-private*, making the remaining declarations private.

Use `def` for read-only variables and `var` for modifiable variables. The `GuitarString` class also provides utility functions that play a note (`playNote`) or stop playing a note (`stopNote`). Along with the sound, guitar strings vibrate on and off with `vibrateOn` and `vibrateOff`. These functions implement the behavior of the `GuitarString` class.

Listing 2.1 Class `GuitarString`—Properties, Variables, and Functions

```

package guitartuner;

import javafx.animation.Interpolator;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.scene.Cursor;
import javafx.scene.CustomNode;
import javafx.scene.Group;
import javafx.scene.input.MouseButton;
import javafx.scene.input.MouseEvent;
import javafx.scene.Node;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import noteplayer.SingleNote;

public class GuitarString extends CustomNode {

```

```
// read-only variables
def stringColor = Color.WHITESMOKE;
// "Strings" are oriented sideways, so stringLength is the
// Rectangle width and stringSize is the Rectangle height
def stringLength = 300;
def stringSize = 1;
def stringMouseSize = 15;
def timeline = Timeline {
    repeatCount: Timeline.INDEFINITE
    autoReverse: true
    keyFrames: [
        at (0s) { vibrateH => 1.0 }
        at (.01s) { vibrateH => 3.0 tween Interpolator.LINEAR }
        at (0s) { vibrateY => 5.0 }
        at (.01s) { vibrateY => 4.0 tween Interpolator.LINEAR }
    ]
};

// properties to be initialized
public-init var synthNote: SingleNote;
public-init var note: Integer;
public-init var yOffset: Number;
public-init var noteText: String;

// class variables
var vibrateH: Number;
var vibrateY: Number;
var play: Rectangle;

function vibrateOn(): Void {
    play.visible = true;
    timeline.play();
}
function vibrateOff(): Void {
    play.visible = false;
    timeline.stop();
}
function playNote(): Void {
    synthNote.noteOn(note);
    vibrateOn();
}
function stopNote(): Void {
    synthNote.noteOff(note);
    vibrateOff();
}
```

Listing 2.2 shows the second part of the code for the `GuitarString` class.

Every class that extends `CustomNode` must define a function `create` that returns a `Node` object.³ Often the node you return will be a `Group`, since `Group` is the most general `Node` type and can include subnodes. But, you can return other `Node` types, such as `Rectangle` (`Shape`) or `HBox` (horizontal box) layout node.

The scene graph for `GuitarString` is interesting because it actually consists of three `Rectangle` nodes and a `Text` node. The first `Rectangle`, used to detect mouse clicks, is completely translucent (its `opacity` is 0). This `Rectangle` is wider than the guitar string so the user can more easily select it with the mouse. Several properties implement its behavior: property `cursor` lets a user know the string is selected and property `onMouseClicked` provides the event handling code (play the note and vibrate the string).

The second `Rectangle` node defines the visible string. The third `Rectangle` node (assigned to variable `play`) “vibrates” by both moving and changing its height. This rectangle is only visible when a note is playing and provides the vibration effect of “plucking” a string. The movement and change in height are achieved with animation and binding. The `Text` node simply displays the letter (E, A, D, etc.) associated with the guitar string’s note.

Listing 2.2 Scene Graph for `GuitarString`

```
protected override function create(): Node {
    return Group {
        content: [
            // Rectangle to detect mouse events for string plucking
            Rectangle {
                x: 0
                y: yOffset
                width: stringLength
                height: stringMouseSize
                // Rectangle has to be "visible" or scene graph will
                // ignore mouse events. Therefore, we make it fully
                // translucent (opacity=0) so it is effectively invisible
                fill: Color.web("#FFFFFF", 0) // translucent
                cursor: Cursor.HAND
                onMouseClicked: function(evt: MouseEvent): Void {
                    if (evt.button == MouseButton.PRIMARY){
                        Timeline {
                            keyFrames: [
                                KeyFrame {
```

3. Well, almost. If you don’t define function `create`, then you must declare the class `abstract`. The `Piano` example (see “Project Piano” on page 167) uses an abstract class.

```

        time: 0s
        action: playNote
    }
    KeyFrame {
        time: 2.8s
        action: stopNote
    }
] // keyFrames

        }.play(); // start Timeline
    } // if
}
} // Rectangle
// Rectangle to render the guitar string
Rectangle {
    x: 0.0
    y: 5 + yOffset
    width: stringLength
    height: stringSize
    fill: stringColor
}
// Special "string" that vibrates by changing its height
// and location
play = Rectangle {
    x: 0.0
    y: yOffset
    width: stringLength
    height: bind vibrateH
    fill: stringColor
    visible: false
    translateY: bind vibrateY
}
Text { // Display guitar string note name
    x: stringLength + 8
    y: 13 + yOffset
    font: Font {
        size: 20
    }
    content: noteText
}
} // Group
} // GuitarString

```

Listing 2.3 shows the code for **Main.fx**, the main program for GuitarTuner.

Listing 2.3 Main.fx

```

package guitartuner;
import guitartuner.GuitarString;
import javafx.scene.effect.DropShadow;
import javafx.scene.Group;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Stop;
import javafx.scene.Scene;
import javafx.scene.shape.Line;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import noteplayer.SingleNote;

def noteValues = [ 40,45,50,55,59,64 ]; // numeric value required by midi
def guitarNotes = [ "E","A","D","G","B","E" ];
// guitar note name
def numberFrets = 2;
def numberStrings = 6;
var singleNote = SingleNote{};
singleNote.setInstrument(27); // "Clean Guitar"

var scene: Scene;
var group: Group;
Stage {
  title: "Guitar Tuner"
  visible: true
  scene: scene = Scene {
    fill: LinearGradient {
      startX: 0.0
      startY: 0.0
      endX: 0.0
      endY: 1.0
      proportional: true
      stops: [
        Stop {
          offset: 0.0
          color: Color.LIGHTGRAY
        },
        Stop {
          offset: 1.0
          color: Color.GRAY
        }
      ]
    }
  ]
  width: 340
  height: 200
  content: [
    group = Group {
      // Center the whole group vertically within the scene
      layoutY: bind (scene.height - group.layoutBounds.height) /

```


Index

A

- abstract class 78
 - example using 169
- access modifiers 88–89
 - default 89
 - primary (*Table*) 89
 - variable (*Table*) 89
- action KeyFrame property 184, 206
- adjustingSelection TextBox property 130
- after with insert keyword 57
- Alert popup dialog 134
- and operator 49
- animation 33–36, 206–218
 - and binding 216–218
 - basics 208
 - Chutes and Ladders example 237–247
 - FadeTransition 226, 229, 260
 - interpolation 209
 - Interpolator.DISCRETE 34, 211
 - Interpolator.EASEBOTH 34, 211
 - Interpolator.EASEIN 34, 211
 - Interpolator.EASEOUT 34, 211
 - Interpolator.LINEAR 34, 211
 - ParallelTransition 226, 231
 - PathTransition 226, 233–237
 - PauseTransition 226
 - Photo Carousel example 273–283
 - RotateTransition 226, 228
 - ScaleTransition 226, 227
 - SequentialTransition 226
 - timeline actions 218–225
 - transitions 225–237
 - TranslateTransition 226, 227
 - translateX Node property 34
 - translateY Node property 34
 - AnimationPath class 236
 - API Key for Flickr 301
 - Arc 108
 - arcHeight Rectangle property 29
 - ArcTo 117, 118, 235
 - sweepFlag property 117
 - arcWidth Rectangle property 29
 - arguments, passing to functions 61
 - arguments, program 64
 - arithmetic operators 48
 - as operator 80, 176
 - assignment operators 48
 - asynchronous HTTP requests 296–300
 - at with KeyFrame objects 213
 - autoReverse Timeline property 33, 207, 210

B

- background loading of images 256
- backgroundLoading Image property 250
- before with insert keyword 57
- Behavior class 153
- bidirectional binding 53
- binding 31–32
 - bound functions 63, 147
 - example with object property 173
 - expressions 52
 - function calls 63
 - object literals 73
 - object property 176
 - on replace 86
 - overriding bind 75
 - sequences 60
 - with animation 216–218
- Blend effect 263
- block expression 50
- blocksMouse Node property 99
 - example 182
- Bloom effect 263
- Boolean type 44
- bound function 63
 - example using 147
- bounding rectangles 125
 - (*Table*) 126
 - positioning with 126
- Bounds 125
 - properties (*Table*) 125
- boundsInLocal Node property 126
- boundsInParent Node property 126
- boundsInScene Node property 126
- break keyword 52
- Button UI component 131
- Byte type 46

C

- canSkip KeyFrame property 207
- Cascading Style Sheets, *See* CSS
- casting expression 80, 176
- catch keyword 91
- centering, dynamic 32
- character strings (Java) 72
- Character type 46
- charAt Java function 73
- CheckBox UI component 131
- ChoiceDialog class (Control) 160

ChoiceDialog custom skinnable UI component 158
 ChoiceDialogSkin class (Skin) 160–164
 Chutes and Ladders 237–247
 mobile ready 332–336
 nested scene graph (*Figure*) 245
 Circle 20, 107
 class instance variables, scope of 87
 ClassCastException and `as` operator 80
 classes 67–75
 abstract 78
 init blocks 70
 instance function 68
 instance variable 68
 mixin inheritance 81–83
 null value 72
 object literal expression 68
 overriding bind 75
 overriding functions 76
 postinit blocks 78
 public-init access modifier 69
 public-read access modifier 70
 super keyword 76
 this keyword 71
 clip Node property 97
 example using 144
 close button 116
 ClosePath 117, 235
 closures 65
 Color class 28
 color 28
 rgb 28
 web 28
 Color.color 28
 Color.rgb 28
 Color.TRANSPARENT 28
 Color.web 28
 ColorAdjust effect 263
 columns TextBox property 130
 ComboBox UI component 131
 Common Elements 316
 concatenate Strings 20, 62
 content Group property 16
 continue keyword 52
 Control class 153
 create CustomNode function 23, 38, 103, 171
 createFromPath AnimationPath function 236
 createFromShape AnimationPath function 236
 CROSSHAIR Cursor type 102
 CSS 148–152
 CSS file for TextButton 156
 CubicCurve 110
 CubicCurveTo 117
 curly braces
 block expressions 50
 embed expressions in Strings 45
 currentRate Timeline property 207
 Cursor class 101
 common types (*Table*) 102
 resize types (*Table*) 102
 cursor Node property 38, 97
 CustomNode class 23, 103, 169
 create function 23, 38, 103, 171
 extending 22, 170
 Photo example 268

D

DateTime class 219
 declarative approach 22
 declarative language 14
 def keyword 19, 43
 (*Tip*) 44
 default access modifier 89
 DEFAULT Cursor type 102
 default mobile emulator 318
 delete keyword 58
 formats of (*Table*) 58
 deleteCharAt Java function 73
 Desktop extensions 316
 Digital Clock Example 219
 DIR pseudo variable 47
 disable Node property 99
 disabled Node property 99
 dot TextBox property 130
 double quotation marks, with String 45
 Double type 46
 dragging, mouse 270–273
 dragX MouseEvent property 270
 dragY MouseEvent property 270
 DropShadow effect 30, 263
 Duration type 33, 45

E

Eclipse IDE 3
 effect
 (*Table*) 263
 Blend 263
 Bloom 263
 ColorAdjust 263
 DropShadow 30, 263
 Flood 263
 GaussianBlur 263
 Glow 263
 InnerShadow 263
 Lighting 263
 MotionBlur 263
 PerspectiveTransform 260, 263
 Reflection 260, 263
 SepiaTone 260, 263
 effect Node property 97
 Ellipse 108
 else keyword 51
 emulator, mobile 317
 equal to operator 49
 error Image property 250
 event
 key 32
 mouse 32
 Examples
 Chutes and Ladders 237–247
 Chutes and Ladders (Mobile) 332–336
 Digital Clock 219
 Flickr Interesting 301–308
 Flickr Interesting (Mobile) 327–332
 Flickr Tag 308–311
 Flickr User 312–314
 GuitarTuner 14–19, 36–41
 Order Your Pizza 142–148
 Order Your Pizza 2 158–165

- Photo Carousel 273–283
- Photo Wall 264–269
- Photo Wall Drag 270–273
- Piano 194–204
- Progress Bar 220–225
- exception handling 91–93
- expressions 50–54
 - bidirectional binding 53
 - binding 52
 - block 50
 - for 51
 - if 51
 - object literals 18
 - while 52
- extends keyword 75

F

- F3 language 1
- FadeTransition 190, 226, 229, 260
- false keyword 44
 - FILE_ pseudo variable 47
- FileInputStream Java class 289
- fill Shape property 20, 28, 104
- finally keyword 91
- fitHeight ImageView property 253
- fitWidth ImageView property 253
- Flickr API Key 301
- Flickr API web service calls 301
- Flickr Interesting 301–308
 - mobile ready 327–332
- Flickr Tag 308–311
- Flickr User 312–314
- Flickr, construct URL for images 302
- Float type 46
- floating-point values 45
- Flood effect 263
- Flow layout component 119, 121
 - example using 187
- focusable Node property 99
- focused Node property 99
- font Text property 115
- for expression 51, 56
- Form Follows Function 1
- formatting with print, println 47
- from with delete keyword 58
- function 61–67
 - binding 63
 - bound 63
 - closures 65
 - defining 61
 - overriding 76
 - overriding in mixins 82
 - passing arguments 61
 - return values 62
 - run 64
 - scope of 87
 - sequence argument (*Tip*) 62
 - types 65–67
- function literal 65
- function types 65–67
- FX.getArguments (*Tip*) 65
- FX.getProperty function 319

G

- GaussianBlur effect 263
- geometry
 - Bounds 125
 - Point2D 124
 - Rectangle2D 125
- GET HttpRequest operation 297
- Glow effect 263
- gradients 26–28
 - LinearGradient fill 26
 - RadialGradient fill 27
- graphical node 15
- greater than (or equal to) operator 49
- Group class 15, 102
- GuitarTuner 14–19
 - hierarchical scene graph (*Figure*) 17
 - nested scene graph (*Figure*) 15
 - source code 36–41

H

- h (hour time unit literal) 45
- HAND Cursor type 102
- HBox layout component 15, 24, 119, 120
 - example using 187
- height Image property 250
- hexadecimal values 45
- HLineTo 117
- hover Node property 99
- HttpRequest class 296–300
 - callbacks for read operations (*Table*) 297
 - GET operation 297
 - start function 298
- Hyperlink UI component 131

I

- id Node property 98, 150
- if expression 51
- Image class 250–252
 - properties (*Table*) 250
 - background loading (*Tip*) 256
 - example using 307
- image ImageView property 253
- ImageView 252–262
 - properties (*Table*) 253
- imperative approach 22
- import statement 24
 - with NetBeans 25
 - with script files 90
- indexOf operator 56, 57
- inheritance 2275–83
 - abstract class 78
 - as operator 80, 176
 - extends keyword 75
 - instanceof operator 80
 - mixee 81
 - mixins 81–83
 - overriding functions 76
 - postinit blocks 78
 - super keyword 76
 - variables and mixins 82
- init blocks 70

InnerShadow effect 263
 insert keyword 22, 57
 instance function 68
 scope of 87
 instance variable 68
 instanceof operator 50, 80
 instant DateTime property 219
 Integer type 45
 interpolation 209
 Interpolator.DISCRETE 34, 211
 Interpolator.EASEBOTH 34, 211
 Interpolator.EASEIN 34, 211
 Interpolator.EASEOUT 34, 211
 Interpolator.LINEAR 34, 211
 into with insert keyword 57
 inverse, bidirectional binding 54
 isInitialized function 85

J

Java API method, calling a 22
 Java character strings 72
 Java Network Launching Protocol 9
 Java objects 72
 Java StringBuffer object 73
 Java types 46
 Java Web Start 9
 JavaFX
 downloading software 3
 execution models 9
 Java Web Start 9
 platform (*Figure*) 2, 316
 Production Suite 3
 SDK 2
 JavaFX keywords (*Table*) 93
 JavaFX Production Suite 3
 JavaFX SDK 2
 JNLP 9
 JSON parsing 293–296

K

key events 32
 KeyCode class 324
 mobile environment 323–326
 requestFocus Node function 100, 325
 KeyCode class 324
 KeyFrame class 33, 35
 action property 184
 properties (*Table*) 206
 time property 185
 keyFrames Timeline property 207
 keywords
 (*Table*) 93
 escaping 93

L

layout components 15, 24119–124
 (*Table*) 119
 Flow 119, 121
 HBox 119, 120
 Stack 119, 123

Tile 119, 122
 VBox 119, 120, 183
 layoutBounds Node property 126
 layoutX Node property 98
 layoutY Node property 98
 less than (or equal to) operator 49
 Lighting effect 263
 Line 110
 line cap styles 105
 LinearGradient fill 26, 171
 proportional property 27
 LineTo 117, 235
 ListView UI component 131
 local variables
 in object literal expression 68
 scope of 87
 logical operators 49
 Long type 46

M

m (minutes time unit literal) 45
 mark TextBox property 130
 milliseconds time unit literal 45
 mixee, mixin inheritance 81
 mixin inheritance 81–83
 overriding functions 82
 overriding variables 82
 variables 82
 mixin keyword 81
 mobile emulator 9
 run application in (NetBeans) 317
 types of 318
 mobile environment
 limitations (*Table*) 317
 changes to Stage 322
 Chutes and Ladders example 332–336
 detecting orientation changes 321, 335
 Flickr Interesting example 327–332
 FX.getProperty function 319
 key events 323–326
 mouse events 323–326
 PROFILE pseudo variable 319
 screen dimensions 320
 Mobile extensions 316
 mod operator 48
 MotionBlur effect 263
 mouse dragging 270–273
 mouse events 32, 101
 mobile environment 323–326
 MouseButton.PRIMARY 33
 MouseEvent class 33
 dragX 270
 dragY 270
 move an object, animation 208
 MOVE Cursor type 102
 MoveTo 117, 235
 ms (milliseconds time unit literal) 45
 mutual exclusion selection behavior 131, 141

N

negation operator 49
 nesting sequences 55

- NetBeans IDE 3
 - compile and run 8
 - edit source code 7
 - generate `import` statements 25
 - mobile emulator 9
 - new project 4
 - project properties 10
 - project, create 4
 - run application in mobile emulator 317
 - run the Main project 8
- new operator with Java objects 72
- Node 15, 2096–103
 - effects (*Table*) 263
 - mouse/keyboard event properties (*Table*) 99
 - rendering properties (*Table*) 97
 - key events 32
 - mouse events 32
 - relative order 16
 - `requestFocus` function 100, 325
 - `toBack` function 16
 - `ToFront` function 16
- `not` (logical complement) operator 49
- `not equal to` operator 49
- `null` value 72
- Number type 45

O

- object literal expressions 18
- object literals 18, 68
 - advantages of 69
 - binding 73
 - local variables (*Tip*) 68
- object reference 62, 68
- octal values 45
- Oliver, Chris 1
- `on` `replace` keywords 83
- `onKeyPressed` Node property 99
- `onKeyReleased` Node property 99
- `onKeyTyped` Node property 99
- `onMouseClicked` Node property 32, 38, 99, 101
- `onMouseDragged` Node property 99, 270
- `onMouseEntered` Node property 99
- `onMouseExited` Node property 100
- `onMouseMoved` Node property 100
- `onMousePressed` Node property 100, 270
- `onMouseReleased` Node property 100, 270
- `onMouseWheelMoved` Node property 100
- `opacity` Node property 97
- operators 47–50
 - `and` 49
 - arithmetic 48
 - `as` 80, 176
 - assignment 48
 - `equal to` 49
 - greater than (or equal to) 49
 - `indexOf` 56, 57
 - `instanceof` 50
 - less than (or equal to) 49
 - logical 49
 - `mod` 48
 - negation 49
 - `not` 49
 - `not equal to` 49
 - `or` 49

- relational 49
- remainder 48
- `reverse` 55
- `sizeof` 55
- `or` operator 49
- Order Your Pizza Swing Example 142–148
 - with custom `ChoiceDialog` component 158–165
- `orientation` `PathTransition` property 235
- `orientation` with mobility 321, 335
- `OrientationType.ORTHOGONAL_TO_TANGENT` 235
- origin, geometric 24
- `overline` Text property 115
- overriding `bind` 75
- overriding functions 76
 - `mixin inheritance` 82
- overriding variables, `mixin inheritance` 82

P

- `package` access modifier 89
- `package` statement 24, 90
- packages 89–91
- `ParallelTransition` 226, 231
- `parse` `PullParser` function 290
- Path 116–119, 234
- Path Element
 - (*Table*) 117
 - and animation 234
 - `ArcTo` 117, 118, 235
 - `ClosePath` 117, 235
 - `CubicCurveTo` 117
 - `HLineTo` 117
 - `LineTo` 117, 235
 - `MoveTo` 117, 235
 - `QuadCurveTo` 117
 - `VLineTo` 117
- `PathTransition` 226233–237
 - `AnimationPath` class 236
 - `orientation` property 235
- `pause` Timeline function 208
- `paused` Timeline property 207
- `PauseTransition` 226
- periodic updates 219, 220
- `PerspectiveTransform` effect 260, 263
- Photo Carousel 273–283
 - nested scene graph (*Figure*) 275
- Photo Wall 264–269
 - nested scene graph (*Figure*) 265
- Photo Wall Drag 270–273
- Piano
 - hierarchical scene graph (*Figure*) 190
 - `PianoKey` class hierarchy (*Figure*) 169
 - source code 194–204
- `PianoKey` class 170
- `placeholder` Image property 250
- `play` Timeline function 208
- `playFromStart` Timeline function 208
- `Point2D` 124
- Polygon 109
- Polyline 111
- Popup component 134
- `postdecrement` operator (`--`) 48
- `postincrement` operator (`++`) 48
- `postinit` blocks 78

- predecrement operator (- -) 48
- predicate, generate sequences 59
- preincrement operator (++) 48
- preserveRatio Image property 250, 252
- preserveRatio ImageView property 253
- press-drag-release mouse gesture 270
- pressed Node property 99
- primary access modifiers (*Table*) 89
- print function 46
 - formatting 47
- printing sequences 55
- println function 46
 - formatting 47
- __PROFILE__ pseudo variable 47
 - using to detect mobility 319
- program arguments 64
- progress bar 220
- Progress Bar Example 220–225
- progress Image property 250
- ProgressBar UI component 132
- ProgressIndicator UI component 132
- promptText TextBox property 130
- properties 71
- proportional LinearGradient property 27
- protected access modifier 89
- pseudo variables 47, 319
- public access modifier 89
- public functions and run 88
- public variables and run 88
- public-init access modifier 36, 69, 89, 170
- public-read access modifier 70, 89
- PullParser class 285–296
 - properties (*Table*) 287
 - FileInputStream Java class 289
- parse function 290

Q

- QuadCurve 109
- QuadCurveTo 117
- qwerty mobile emulator 318, 319

R

- RadialGradient fill 27, 172
- RadioButton UI component 131
- rate Timeline property 207
- rawText TextBox property 130
- Rectangle 20, 107
 - arcHeight property 29
 - arcWidth property 29
 - rounded corners 29, 191
- Rectangle2D 125
- Reflection effect 260, 263
- relational operators 49
 - sequence comparison 59
- remainder operator 48
- repeatCount Timeline property 207, 210
- replace Java function 73
- replace keyword (with on) 83
- requestFocus Node function 100, 325
- RESTful web service call 301
- return values from functions 62
- reverse operator 55

- rotate Node property 98, 261
- Rotate transformation 334
- RotateTransition 226, 228
- rotating in response to orientation changes 335
- rounded rectangle 29, 191
- run function 64, 329
- running Timeline property 207

S

- s (seconds time unit literal) 45
- Scalable Vector Graphics, *See* SVG
- ScaleTransition 226, 227
- scaleX Node property 98
- scaleY Node property 98
- scaling 256–259
- Scene 96
- scene graph 15, 16
- scope
 - class instance variables 87
 - instance functions 87
 - local variables 87
 - script functions 87
 - script variables 87
- script files 88–91
- script functions, scope of 87
- script variables, scope of 87
- script-level variables 25
- script-private access 36
- ScrollBar UI component 131
- SepiaTone effect 260, 263
- sequence 21–22, 54–60
 - accessing items 57
 - binding 60
 - comparison 59
 - delete keyword 58
 - example 182
 - for expression 56
 - formats of delete keyword (*Table*) 58
 - indexOf operator 56, 57
 - insert keyword 22, 57
 - literal 54
 - nesting 55
 - of values 54
 - on replace 84
 - predicates 59
 - printing 55
 - reverse operator 55
 - sizeof operator 55
 - slices 59
 - step in range literal 55
 - where clause 56
- sequence literal 21, 54
- SequentialTransition 226
- setCharAt Java function 73
- Shape 20103–116
 - common properties (*Table*) 104
 - Arc 108
 - Circle 107
 - CubicCurve 110
 - Ellipse 108
 - line cap styles 105
 - Lines 110
 - Polygon 109
 - Polyline 111

- QuadCurve 109
- Rectangle 107
- ShapeIntersect 113
- ShapeSubtract 113
- SVGPath 111
- Text 115
- ShapeIntersect 113
- ShapeSubtract 113
- Short type 46
- single quotation marks, with String 45
- sizeof operator 55
- Skin class 153
- skinnable UI component 148–152
 - Behavior class 153
 - Control class 153
 - custom 152, 158
 - Skin class 153
- slices, sequence 59
- slide show, animated 274
- Slider UI component 131
- smooth Image property 250
- smooth ImageView property 253
- smooth Shape property 104
- Stack layout component 119, 123
- Stage 15, 95
 - dimensions and mobility 320
 - orientation changes 322
- StageStyle.DECORATED 95
- StageStyle.UNDECORATED 96
- start HttpRequest function 298
- static typing 14
- step in range literal 55
- stop Timeline function 208
- strikethrough Text property 115
- String type 20, 45
 - { } 20, 45
 - concatenate 20, 62
 - curly braces 45
 - double quotation marks 45
 - single quotation marks 45
- StringBuffer object (Java) 73
- stroke Shape property 20, 104
- strokeDashArray Shape property 101, 104, 106
- strokeDashOffset Shape property 104
- strokeLineCap Shape property 104
- strokeLineJoin Shape property 104
- strokeMiterLimit Shape property 104
- strokeWidth Shape property 20, 104
- style Node property 98
- styleClass Node property 98, 150
- stylesheets Scene property 151
- super keyword 76
- SVG 3
- SVG file 3
- SVGPath 111
- sweepFlag ArcTo property 117
- Swing components 135–142
 - example application 142–148
 - SwingButton 135, 187, 260
 - SwingCheckBox 135
 - SwingComboBox 136
 - SwingComboBoxItem 136
 - SwingIcon 137
 - SwingLabel 137
 - SwingList 138
 - SwingListItem 138

- SwingRadioButton 140
- SwingScrollPane 138
- SwingSlider 141
- SwingTextField 141
- SwingToggleButton 140
- SwingToggleGroup class 141
- SwingButton component 135, 260
 - example using 187
- SwingCheckBox component 135
 - example using 146
- SwingComboBox component 136
- SwingComboBoxItem component 136
- SwingIcon component 137
- SwingLabel component 137
- SwingList component 138
- SwingListItem component 138
- SwingRadioButton component 140
 - example using 145
- SwingScrollPane component 138
- SwingSlider component 141
- SwingTextField component 141
 - example using 145
- SwingToggleButton component 140
- SwingToggleGroup class 141

T

- Text 115
 - properties (*Table*) 115
 - close button 116
- TEXT Cursor type 102
- textAlignment Text property 115
- TextBox UI component 129–131
 - properties (*Table*) 130
 - example using 310
- TextButton class (Control) 153
- TextButton custom skinnable UI component 152
 - CSS file 156
- TextButtonSkin class (Skin) 155
- textOrigin Text property 116
- this keyword 71
- throw keyword 92
- Tile layout component 119, 122
- time KeyFrame property 185, 206
- time Timeline property 207
- Timeline class 33206–218
 - functions (*Table*) 208
 - interpolators (*Table*) 211
 - properties (*Table*) 207
 - actions 218–225
 - check running status (*Tip*) 224
 - pause function 33, 208
 - play function 33, 208
 - playFromStart function 33, 208
 - stop function 33, 208
 - stop itself (*Tip*) 223
- timelines KeyFrame property 207
- toBack Node function 16
- toFront Node function 16
- ToggleButton UI component 131
- ToggleGroup class 131
- touch mobile emulator 318
- transforms Node property 256–259
- transforms Node property 98, 334
- transitions 225–237

- (Table)* 226
- basics 226
- TranslateTransition 226, 227
- translateX Node property 34, 98
 - animation 209
- translateY Node property 34, 98
- triggers 83–86
 - binding 86
 - isInitialized function 85
 - on replace example 306
 - on replace keywords 83
 - sequences 84
 - variables 83
- trim Java function 73
- true keyword 44
- try keyword 91
- TV extensions 316
- tween key frame operator 34, 209
- two-dimensional coordinate system 24
- type inference 14, 19, 44
- types 44–46
 - Boolean 44
 - Byte 46
 - Character 46
 - Double 46
 - Duration 45
 - Float 46
 - Integer 45
 - Java wrapper types 46
 - Long 46
 - Number 45
 - Short 46
 - String 45
 - Void 46, 61

U

- UI components
 - (Table)* 131
 - Alert popup dialog 134
 - Button 131
 - CheckBox 131
 - custom ChoiceDialog component 158
 - custom TextButton component 152
 - Hyperlink 131
 - ListView 131
 - Popup window 134
 - ProgressBar 132
 - ProgressIndicator 132
 - RadioButton 131
 - ScrollBar 131
 - skin with CSS 148–152
 - Slider 131
 - TextBox 129–131
 - ToggleButton 131
 - ToggleGroup class 131
- underline Text property 115
- url Image property 250

V

- values KeyFrame property 206
- var keyword 19, 43
- variable 1943–44

- access modifiers *(Table)* 89
 - on replace 83
 - overriding in mixins 82
 - scope of 87
 - script-level 25
- VBox layout component 15, 24, 119, 120
 - example using 183
- viewport ImageView property 253
- visible Node property 98
- VLineTo 117
- Void type 46, 61

W

- WAIT Cursor type 102
- web services
 - Flick Interesting example 301–308
 - Flick Tag example 308–311
 - Flick User example 312–314
 - Flickr 301
- where clause 56
- while expression 52
- width Image property 250
- windows
 - Alert popup dialog 134
 - Popup component 134
- with inverse, bidirectional binding 54
- wrappingWidth Text property 115

X

- XML parsing 286–292