



Controls, charts, and storage

This chapter covers

- Creating forms using standard controls
- Storing data (even on applets and phones)
- Playing with 3D charts and graphs
- Writing our own skinnable control

There can't be many programmers who haven't heard of the Xerox Alto, if not by name then certainly by reputation. The Alto was a pioneering desktop computer created in the 1970s at the Xerox's Palo Alto Research Center (PARC). It boasted the first modern-style GUI, but today it's probably best remembered as the inspiration behind the Apple Macintosh. Although graphics have become more colorful since those early monochrome days, fundamentally the GUI has changed very little. A time traveler from 1985 (perhaps arriving in a DeLorean sports car?) may be impressed by the beauty of modern desktop software but would still recognize the majority of UI widgets. UI stalwarts like buttons, lists, and check boxes still dominate. The World Wide Web popularized the hypertext pane, but that

aside, very few innovations have really caught on. But then, if something works, why fix it?

But one problem *did* need fixing. The Xerox PARC GUI worked well for a desktop computer like the Alto but was wholly inappropriate for small-screen devices like a cell phone. So mobile devices found their own ways of doing things, creating *little-brother* equivalents to many standard desktop widgets. But, inevitably, little brothers grow up: mobile screens got better, graphics performance increased, and processing power seemed to leap with each new product release. Suddenly phones, media players, games consoles, and set-top boxes started to sport UIs looking uncannily like their desktop siblings, yet independent UI toolkits were still used for each.

One remit of the JavaFX project was to unite the different UI markets—PCs, Blu-ray players, cell phones, and other devices—under one universal toolkit, to create (as Tolkien might have put it) *one toolkit to rule them all*. Java had always had powerful desktop UI libraries (Abstract Windows Toolkit [AWT] and Swing), but they were far too complex for smaller devices. Neither used the *retained-mode* graphics favored by JavaFX’s scene graph, and neither was designed to be constructed declaratively, making them alien to the way JavaFX would like to work.

The JavaFX team planned a new library of UI widgets: universal, lightweight, modern, and easily restyled. The result of their labor was the *controls* API introduced in JavaFX 1.2, designed to complement (and eventually replace) the desktop-only Swing wrappers in `javafx.ext.swing`, which had shipped with earlier JavaFX versions.

In this chapter we’re going to learn how to use the new controls API, along with other features that debuted in 1.2. So far we’ve had a lot of fun with games, animations, and media, but JavaFX’s controls permit us to write more serious applications, so that’s what we’ll be doing in this chapter’s project. Let’s begin by outlining the application we’re building.

7.1 **Comments welcome: Feedback, version 1**

In this chapter we’ll develop a small feedback form using JavaFX controls. Different control types will collect the answers to our questions, a persistent storage API will save this data, and some chart/graph controls will display the resulting statistics (see figure 7.1). Along the way we’ll also expand our knowledge of JavaFX’s container nodes.

There are just two classes in this project, but they pack quite a punch in terms of new material covered. As with previous projects, the code has been broken into versions. Version 1 will focus on building the input form using the new JavaFX 1.2 controls, and version 2 will save the data and create charts from it.

Although we’ll be covering only a subset of the controls available under JavaFX 1.2—buttons, text fields, radio buttons, and sliders—there’s enough variety to give you a taste of the controls library as a whole. We begin not with the controls but with a model class to bind our UI against.

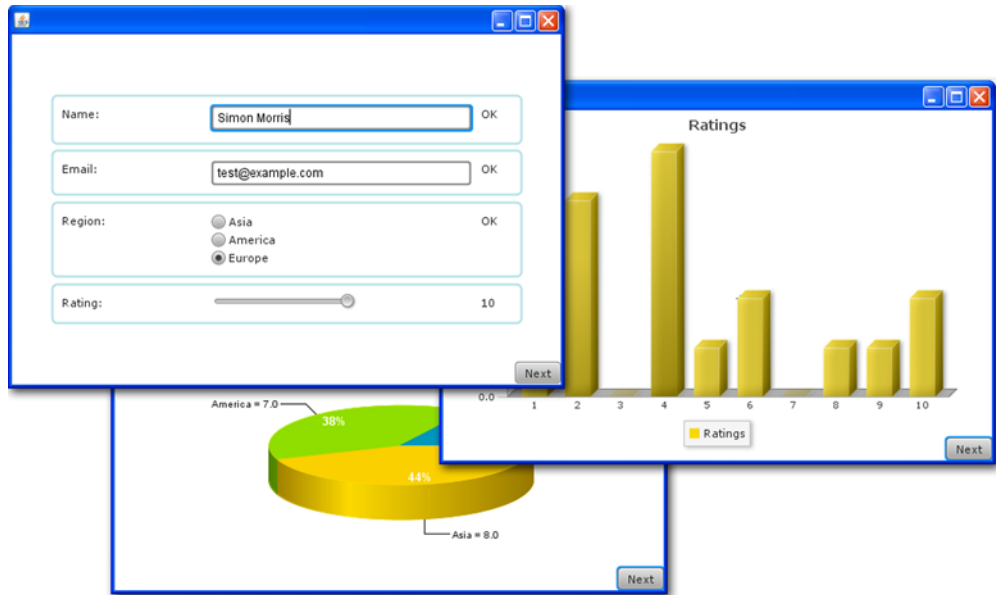


Figure 7.1 A bar chart, with 3D effect, showing feedback scores from contributors

7.1.1 The Record class: a bound model for our UI

To hold the data we're collecting for our form we need a class, such as the one in listing 7.1. Variables store the data for each field on the form, and a set of corresponding booleans reveals whether each value is valid. From a design point of view, it's useful to keep the logic determining the validity of a given field close to its data storage. In *ye olde times* (when AWT and Swing were all we had) this logic would be scattered across several far-flung event handlers, but not with JavaFX Script.

Listing 7.1 Record.fx (version 1)

```
package jfxia.chapter7;

package def REGIONS:String[] = [ "Asia","America","Europe" ];

package class Record {
    package var name:String;
    package var email:String;
    package var region:Integer = -1;
    package var rating:Number = 0;

    package def validName:Boolean = bind (    ← Name valid?
        name.length() > 2
    );
    package def validEmail:Boolean = bind (    ← Email address valid?
        (email.length() > 7) and
        (email.indexOf("@") > 0)
    );
    package def validRegion:Boolean = bind (    ← Region set?
```

```

        region >= 0
    );
    package def validRating:Boolean = bind (    <— Rating set?
        rating > 0
    );
    package def valid:Boolean = bind (    <— All fields valid?
        validName and validEmail and
        validRegion and validRating
    );
}

```

This is our initial data class: a model to wire into our UI’s view/controller—recall the Model/View/Controller (MVC) paradigm. The class consists of four basic feedback items, each with a corresponding Boolean variable bound to its validity, plus a master validity boolean:

- The name variable holds the name of the feedback user, and the `validName` variable checks to see if its length is over two characters.
- The email variable holds the email address of the feedback user, and the `validEmail` variable checks to see if it is at least eight characters long and has an @ (at) symbol somewhere after the first character.
- The region variable stores the location of the user. A sequence of valid region names, `REGIONS`, appears at the head of the code, in the script context. The `validRegion` variable checks to see that the default, `-1`, is not set.
- The rating variable holds a feedback score, between 1 and 10. The `validRating` variable checks to see whether the default, `0`, is not set.

An extra variable, `valid`, is a kind of *master boolean*, depending on the validity of all the other variables. It determines whether the `Record` as a whole is ready to be saved.

This four-field data class is what we’ll base our UI on. Sure, we could ask more than four questions (in the real world we certainly would), but this wouldn’t really teach us anything new. The four we have will be more than enough for this project, but feel free to add your own if you want.

We have a class to store the data; all we need now is a user interface.

7.1.2 The Feedback class: controls and panel containers

Java calls them *components*; I sometimes call them by the generic term *widgets*, but the new (official) JavaFX name for UI elements is *controls*, it seems. Controls are buttons, text fields, sliders, and other functional bits and pieces that enable us to collect input from the user and display output in return. In our feedback form we’ll use a text field to collect the respondents’ name and email address, we’ll use radio buttons to allow them to tell us where they live, and we’ll use a slider to let them provide a score out of 10. Figure 7.2 shows what the interface will look like.

A Next button sits in the corner of the window, allowing the user to move to the next page of the application. In the finished project the button will initially submit the feedback form (assuming all fields are valid) and then become a toggle, so users can

Figure 7.2 Our project's simple feedback form, complete with text fields, radio buttons, and sliders. Oh, and a Next button in the corner!

jump between a bar chart and a pie chart that summarize the data already collected. In version 1, however, the charts aren't available, so we'll just use the button to print the current data in the model object.

Because the UI code is quite long, I've broken it into four smaller listings (listings 7.2-7.5), each dealing with a different part of the interface. The first part, listing 7.2, begins with a long shopping list of classes (many for control and layout) that need to be imported.

Listing 7.2 Feedback.fx (version 1, part 1)

```
package jfxia.chapter7;

import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.RadioButton;
import javafx.scene.control.Slider;
import javafx.scene.control.Textbox;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.Panel;
import javafx.scene.layout.Tile;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.util.Sequences;

def record:Record = Record {};    ← Our data model

def winW:Number = 550;
def winH:Number = 350;
// Part 2 is listing 7.3; part 3, listing 7.4; part 4, listing 7.5
```

At the start of the code we see a new Record object being declared. Recall that this is the model class our UI will plumb itself into. The next two lines define a couple of constants that will be used as Scene dimensions in the part 2 of the code. And speaking of part 2, listing 7.3 is next.

Listing 7.3 Feedback.fx (version 1, part 2)

```
// Part 1 is listing 7.2
def nextButton = Button {
  text: "Next";
  action: function() {
    println("{record.name} {record.email} "
      "{record.region} {record.rating}");
  }
}

var mainPan:Panel;
Stage {
  scene: Scene {
    content: mainPan = Panel {
      prefWidth: function(w:Number) { winW };
      prefHeight: function(h:Number) { winH };
      onLayout: function() {
        mainPan.resizeContent();
        var c = mainPan.content;
        for(n in mainPan.getManaged(c)) {
          def node:Node = n as Node;
          var x = winW-node.layoutBounds.width;
          var y = winH-node.layoutBounds.height;
          if(not (node instanceof Button)) {
            x/=2; y/=2;
          }
          mainPan.positionNode(node , x,y);
        }
      }
      content: [
        createFeedbackUI() ,
        nextButton
      ]
    }
    width: winW; height: winH;
  }
  resizable: false;
}

// Part 3 is listing 7.4; part 4, listing 7.5
```

Button in southeast corner

Just print record details

Declarative custom layout

Called to lay out children

Don't center nextButton

Position node

Create window's scene graph

The Button class creates a standard button control, like the Swing buttons we saw in previous chapters, except entirely scene-graph based and therefore not confined to the desktop. Our nextButton object is very simple: text to display and an action function to run when clicked. In version 2 this button will permit users to move between pages of the application, but for version 1 all it does is print the data in our Record.

The most interesting part of listing 7.3 is the Panel class—clearly it's some kind of scene graph node, but it looks far more complex than the nodes we've encountered

so far. You may recall that when we built the video player project, we created our own custom layout node by extending `javafx.scene.layout.Container`. Because we created a full container class, we could use it over and over in different places. But what if we wanted a one-shot layout? Do we have to go to all the hassle of writing a separate class each time?

No, we don't. The `javafx.scene.layout.Panel` class is a lightweight custom layout node. Its mechanics can be plugged in declaratively, making it ideal for creating one-shot containers, without the pain of writing a whole new class. The class has several variables that can be populated with anonymous functions to report its minimum, maximum, and preferred size, plus the all-important `onLayout` function, for actually positioning its children. Let's refresh our memory of the layout code in listing 7.3.

```
onLayout: function() {
    mainPan.resizeContent();
    var c = mainPan.content;
    for(n in mainPan.getManaged(c)) {
        def node:Node = n as Node;
        var x = winW-node.layoutBounds.width;
        var y = winH-node.layoutBounds.height;
        if(not (node instanceof Button)) { x/=2; y/=2; }
        mainPan.positionNode(node , x,y);
    }
}
```

The `mainPan` variable is a reference to the `Panel` object itself, so `mainPan.resizeContent()` will resize the `Panel`'s own children to their preferred dimensions. The code then loops over each node requiring layout, centering it within the `Scene` (or rather, the constants used to size our `Scene`). Note, however, that we do not center any node of type `Button`—this is why the `Next` button ends up in corner of the window.

At the end of listing 7.3 you'll notice that, aside from the `nextButton`, the contents of the panel are populated by a function called `createFeedbackUI()`. Listing 7.4 shows us its detail.

Listing 7.4 Feedback.fx (version 1, part 3)

```
// Part 2 is listing 7.3; part 1, listing 7.2
function createFeedbackUI() : Node {
    def ok:String = "OK";
    def bad:String = "BAD";

    var togGrp = ToggleGroup {};
    def selected = bind togGrp.selectedButton
    on replace {
        if(togGrp.selectedButton != null) {
            record.region = Sequences.indexOf
                (togGrp.buttons , togGrp.selectedButton);
        }
    }

    VBox {
        var sl:Slider;
```

Function builds
feedback form

RadioButton
group

```

spacing: 4;
content: [
    createRow(      ← Name row
        "Name:" ,
        TextBox {
            columns:30;
            text: bind record.name with inverse;
        } ,
        Label {
            text: bind
                if(record.validName) ok else bad;
        }
    ) ,
    createRow(      ← Email row
        "Email:" ,
        TextBox {
            columns: 30;
            text: bind record.email with inverse;
        } ,
        Label {
            text: bind
                if(record.validEmail) ok else bad;
        }
    ) ,
    createRow(      ← Region row
        "Region:" ,
        Tile {
            columns: 1;
            content: for(r in Record.REGIONS) {
                def idx:Integer = (indexof r);
                RadioButton {
                    text: r;
                    toggleGroup: togGrp;
                    selected: (record.region==idx);
                }
            }
        } ,
        Label {
            text: bind
                if(record.validRegion) ok else bad;
        }
    ) ,
    createRow(      ← Rating row
        "Rating:" ,
        sl = Slider {
            max: 10;
            value: bind record.rating with inverse;
        } ,
        Label {
            text: text: bind if(record.validRating)
                "{sl.value as Integer}" else bad;
        }
    )
]

```

Assign to ToggleGroup


```

    }
}
// Part 4 in listing 7.5

```

Each *page* in the application has its own function to create it. Separating each part of the scene graph into functions makes things more manageable. Listing 7.4 shows the function that creates the feedback form itself. It adds a row for each input field in the form, using a function called `createRow()` that we'll look at in the final listing for this source file. Each row consists of a string to act as the field's label, the control being used to collect the data, and a `Label` to display the validity of the current data.

A set of unfamiliar controls has been introduced in listing 7.4: `TextBox` provides a simple text entry field, `Slider` permits the user to choose a value within a given range (we saw it briefly in the last chapter), and `RadioButton` allows the user to make a choice from a given set of options. (Legend has it radio buttons got their name from the old-fashioned car radios, with mechanical push-button tuning.)

The `RadioButton` controls are added, via a `for` loop, to a `Tile` container, configured to position them in a single row. Because the `RadioButton` controls have to cooperate, they refer to a common `ToggleGroup` object. All `RadioButton` controls added to a given group will work together to ensure only one button is ever selected at any given time. Each grouping of radio buttons in a UI must have its own `ToggleGroup` object to manage it, or the buttons won't be able to cooperate.

Figure 7.3 shows the flow of data that connects the model to the UI. These connections are made through binds. Each control (with the exception of the radio buttons) has a bidirectional bind to its associated data in the model. If the model changes, the control updates; likewise, if the control changes, the model updates. The `Label` at the end of each row is also bound to the model; whenever any data in the model changes, the associated validity boolean updates, and this in turn causes the `Label` to update.

Unfortunately the `ToggleGroup` that controls our radio buttons isn't so easy to bind against. It has a variable to tell us which `RadioButton` is currently selected but not the index of the button—the index is what we really need to know. Figure 7.4 shows the workaround we employ to convert the `RadioButton` into an index. We create a variable (`selected`) that binds against `ToggleGroup`'s `selectedButton` field, running a trigger each time it changes. The trigger translates the selected `ToggleButton` (a superclass of

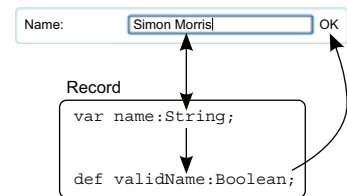


Figure 7.3 The flow of updates between the model and the UI: the data and text box are bidirectionally bound, the validity boolean is bound to the data, and the validity label is bound to the validity boolean.

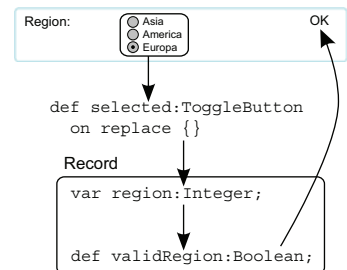


Figure 7.4 To update the model from a `ToggleGroup` we bind a variable against the group's selected button and then use a trigger to translate any changes into the button's index value.

RadioButton) into an index and pushes the value into the model. This causes the familiar validity boolean updates.

Regrettably, because the relationship between control and model is not bidirectional, a change to the model is not automatically reflected in the UI. (Hopefully future enhancements to the ToggleGroup class will make this type of binding easier.)

The final listing for this source code file is presented in listing 7.5. It shows the `createRow()` function we saw producing each row of the feedback form.

Listing 7.5 Feedback.fx (version 1, part 4)

```
// Part 3 is listing 7.4; part 2, listing 7.3; part 1, listing 7.2
function createRow(lab:String,n:Node,v:Node) : Node {
  def margin:Number = 10;

  n.layoutX = 150;          | Position control
  v.layoutX = 420;          | and label

  var grp:Group = Group {
    layoutX: margin;
    layoutY: margin;
    content: [ Label{text:lab} , n , v ]
  }
  Group {
    content: [
      Rectangle {
        fill: null;
        stroke: Color.POWDERBLUE;
        strokeWidth: 2;
        width: 450 + margin*2;
        height: bind
          grp.layoutBounds.height + margin*2;
        arcWidth: 10; arcHeight: 10;
      } ,
      grp
    ]
  }
}
```

Build controls
group

Use group
inside border

This function is a convenience for positioning each row of the feedback form. It accepts a `String` and two `Node` objects: the `String` is turned into a `Label`, and the two nodes are positioned at set points horizontally (so the overall effect is three columns running down the form). Absolute positioning of nodes like this is less flexible than using a layout container but can be worthwhile if the UI is static and would otherwise demand custom layout code. In the sample code the rectangles grouping each row rule out the use of something like the standard `Tile` container. Figure 7.5 shows the relationship between the function parameters and the chunk of scene graph the function creates.

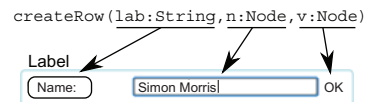


Figure 7.5 The `createRow()` function is a convenience for manufacturing each part of the feedback form. Each row consists of three parts: a text label and two nodes (controls).

The three nodes are gathered inside a Group and then added to another Group that provides a pin-line border by way of a Rectangle node.

The `createRow()` function draws to a close version 1 of the source code for our feedback form. These two simple classes, `Record` and `Feedback`, provide the entirety of the application. Now it's time to test the code.

7.1.3 Running version 1

Building and running version 1 of the project results in the window shown in figure 7.6.

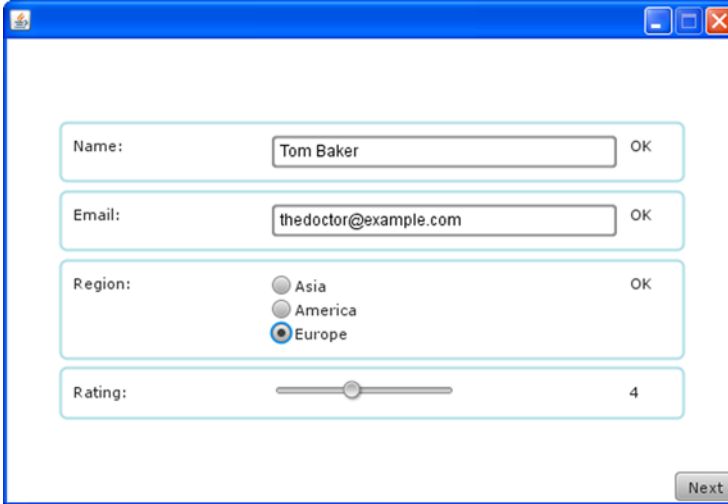
The image shows a JavaFX application window with a blue title bar. Inside, there's a feedback form with four rows. Each row has a label, a control, and an 'OK' button. The first row is 'Name:' with a text box containing 'Tom Baker'. The second row is 'Email:' with a text box containing 'thedoctor@example.com'. The third row is 'Region:' with three radio buttons labeled 'Asia', 'America', and 'Europe'; 'Europe' is selected. The fourth row is 'Rating:' with a slider control set to the value '4'. A 'Next' button is located at the bottom right of the form area.

Figure 7.6
Version 1 of the
application, running

We can edit the four data fields of the form, and their corresponding labels will change to reflect their content's validity. Clicking the Next button merely prints the details in the model, `Record`, to the console. Obviously this isn't particularly useful, unless we intend to deliberately ignore our users' feedback (which, of course, no self-respecting company would ever do!), so in the next part of this chapter we'll store the data from the form and produce some statistics.

7.2 Chart topping: Feedback, version 2

So we have a simple feedback form; it asks only four questions, but that's enough to derive some statistics from. In order to create those statistics, we need to record the details provided by each user. On a desktop application this would be as simple as opening a file and writing to it, but JavaFX applications might need to run on cell phones or inside a web page, where writing to a disk isn't usually an option. Fortunately JavaFX 1.2 provides its own persistent storage solution, which we'll explore in the coming section.

Once the data is stored safely, we need to find something to do with it. And again JavaFX provides an answer, in the form of its chart controls. All manner of graphs and charts can be created from collections of data and displayed in suitably impressive

ways. We'll be using a couple of simple 3D charts to display the region and rating data from our form. So that's the plan; now let's get started.

7.2.1 Cross-platform persistent storage

Almost every application needs to record data—even games like to save their high-score tables. In a thin-client environment, working exclusively against data on a network, we have the luxury of not worrying about the storage capabilities of the host device, but fatter clients aren't so lucky. A problem arises about how to persist data across all the devices JavaFX supports. On the desktop we can fall back to Java's I/O classes to read and write files on the host's hard disk, but what happens if our code finds itself running as an applet, or on a cell phone, or on a Blu-ray player—where can it save its data then?

The `javafx.io` package provides a clean, cross-platform, persistence mechanism for that very purpose. It allows us to read and write files in a managed way across a variety of devices. By *managed*, I mean these files are firmly associated with the running application. In effect, they behave like a cross between the files of a regular desktop application and the cookies of a web page. Let's take a look at a demonstration, by way of listing 7.6.

Listing 7.6 Record.fx (version 2)

```
package jfxia.chapter7;

import javafx.io.Resource;
import javafx.io.Storage;
import javafx.io.http.URLConverter;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintStream;

package def REGIONS:String[] = [ "Asia", "America", "Europe" ];

package class Record {
    package var name:String;
    package var email:String;
    package var region:Integer = -1;
    package var rating:Number = 0;

    package def validName:Boolean = bind (
        name.length() > 2
    );
    package def validEmail:Boolean = bind (
        (email.length() > 7) and
        (email.indexOf("@") > 0)
    );
    package def validRegion:Boolean = bind (
        region >= 0
    );
    package def validRating:Boolean = bind (
        rating > 0
    );
};
```

```

package def valid:Boolean = bind (
    validName and validEmail and
    validRegion and validRating
);

package function encode(r:Record) : String {
    def conv = URLConverter{}
    "{conv.encodeString(r.name)}&"
    "{conv.encodeString(r.email)}&"
    "{r.region}&"
    "{r.rating as Integer}"
}

package function decode(s:String) : Record {
    def conv = URLConverter{}
    def arr:String[] = s.split("&");
    if(sizeof arr < 4) {
        null
    }
    else {
        Record {
            name: conv.decodeString(arr[0]);
            email: conv.decodeString(arr[1]);
            region: Integer.parseInt(
                conv.decodeString(arr[2]));
            rating: Integer.parseInt(
                conv.decodeString(arr[3]));
        }
    }
}

def FILENAME:String = "feedback.txt";
package function load() : Record[] {
    var recs:Record[] = [];

    def data:Storage = Storage { source: FILENAME; }
    def resource:Resource = data.resource;

    if(resource.readable) {
        def br:BufferedReader = new BufferedReader(
            new InputStreamReader(
                resource.openInputStream()
            )
        );
        var in:String = br.readLine();
        while(in!=null) {
            insert Record.decode(in) into recs;
            in = br.readLine();
        }
        br.close();
    }

    recs;
}

package function save(recs:Record[]) : Void {
    def data:Storage = Storage { source: FILENAME; }
    def resource:Resource = data.resource;

```

Encode record into string

Decode record from string

Error: too few fields

Create Record object

Filename for data

Load all records

Use Java's BufferedReader class

Call our decode() function

Save all records

```

if(resource.writable) {
    def ps:PrintStream = new PrintStream(
        resource.openOutputStream(true)
    );
    for(r in recs) {
        ps.println(Record.encode(r));
    }
    ps.close();
}
}

```

Call our encode()
function

This listing adds a mass of code to load and save the `Record` objects our UI populates. At the head of the file we see six I/O-related classes being imported. The first three are from `JavaFX` and the final three are from `Java`. (I'll explain why we need the `Java` classes later.)

Below the `Record` class itself (unchanged from the last version) we find four brand-new script-level functions. The first two are named `encode()` and `decode()`, and they translate our data to and from a `String`, using the `javafx.io.http.URLConverter` class. This class provides useful functions for working with web addresses, specifically encoding and decoding the parameters on HTTP GET requests (when query data is encoded as part of the URL). The `encode()` function uses `URLConverter` to turn a `Record` object into a `String`, so whitespace and other potentially troublesome characters are taken care of. Its companion reverses the process, reconstructing a `Record` from an encoded `String`.

The second two functions in listing 7.6, named `load()` and `save()`, make use of this encode/decode functionality to store and later re-create an entire bank of `Record` objects to a file. To locate the file we create a `Storage` object with the required filename and use its embedded `Resource` object to work with the data. Let's remind ourselves of the body of the `load()` function:

```

def data:Storage = Storage { source: FILENAME; }
def resource:Resource = data.resource;
if(resource.readable) {
    def br:BufferedReader = new BufferedReader(
        new InputStreamReader(
            resource.openInputStream()
        )
    );
    var in:String = br.readLine();
    while(in!=null) {
        insert Record.decode(in) into recs;
        in = br.readLine();
    }
    br.close();
}

```

Variables in the `Resource` class allow us to check on the state of the file (for example, its readability) and access its input stream. In the example, we use `Java`'s `BufferedReader`, wrapped around an `InputStreamReader`, to pull data from the input stream as text. `Java`'s reader classes know how to interpret different character encodings and

deal with non-English characters (like letters with accents). The `decode()` function we saw earlier is then used to translate this text into a `Record` object.

Using Java's I/O classes

It's a shame we have to use Java's reader/writer classes, as we can't be sure those classes will be available outside the desktop environment. JavaFX 1.2 doesn't yet have a full complement of its own readers and writers, so for the time being we either have to handle the raw byte stream ourselves, use JavaScript Object Notation (JSON)/eXtensible Markup Language (XML), or revert to Java's I/O classes.

Now let's turn our attention to the body of the `save()` function:

```
def data:Storage = Storage { source: FILENAME; }
def resource:Resource = data.resource;
if(resource.writable) {
    def ps:PrintStream = new PrintStream(
        resource.openOutputStream(true)
    );
    for(r in recs) {
        ps.println(Record.encode(r));
    }
    ps.close();
}
```

Again `Storage` and `Resource` are used to establish a link to the data, and Java classes are used to write it, combined with the `encode()` function to turn each `Record` into a `String`.

Using JavaFX's persistence API we can save and recover data to the host device, regardless of whether it's a phone, a games console, or a PC. Although the entry point into the data is described as a filename, we cannot use the API to access arbitrary files on the host. A bit like cookies on a website, the mechanism links data to the application that wrote it, a process that deserves a little further explanation.

7.2.2 How Storage manages its files

The `Storage` class segregates the files it saves using the application's URL domain and path. In listing 7.6 we saw the `Storage` class being used to access a file with a basic filename, `feedback.txt`, but it's also possible to prefix a filename with a path, like `/JFX/FeedbackApplet/feedback.txt`. To explain what impact this has on how the data is stored and accessed, we need to look at some example web addresses:

- <http://www.jfxia.com/Games/PacmanGame/index.html>
- <http://www.jfxia.com/Games/SpaceInvaders/index.html>

For JavaFX code from the above URLs, the persistence API would create an original space for the www.jfxia.com domain within its storage area (exactly where this is doesn't concern us; indeed it might be different for each device and/or manufacturer). This

ensures files associated with applets/applications from different organizations or authors are kept apart. Within that space, files can be further managed by using the path element of the URL. The location of the HTML, JAR, or WebStart JNLP file (depending on what is being referenced by the URL) is taken to be the default directory of the application, and permission is granted to read or write files using that directory path or any path that's a direct or indirect parent or descendant.

Let's decipher that rather cryptic last sentence by using the two example URLs.

- If the applet at `/Games/PacmanGame/index.html` used the `Storage` class to create a file called `scores.dat`, the persistence API would write it into `/Games/PacmanGame/scores.dat`, within its area set aside for the www.jfxia.com domain. Because the applet didn't specify a path prefix for the filename, the API used the path of the HTML page the applet was on.
- The applet could alternatively specify the absolute path, `/Games/PacmanGame/scores.dat`, resulting in the same effect as `scores.dat` on its own.
- The applet could also have specified a subdirectory off of its *home directory*, such as `/Games/PacmanGame/Data/scores.dat`. As a descendant of the home directory, it would be granted permission for reading and writing. (Note: no such physical directory has to exist on the www.jfxia.com web server. These file paths are merely data source identifiers, and the data isn't being stored on the server side anyway!)
- The applet could alternatively have used the `/Games/` or `/` (root) directory. Because both are parents of the applet's home directory, they would also have been permitted. But here's the important part: while both `/Games/SpaceInvaders/index.html` and `/Games/PacmanGame/index.html` can access the root and the Games directory, they cannot access each other's directories or any subdirectories therein.

In other words, if either game writes its data *above* its home directory, that data is available to other applications living (directly or indirectly) off that directory. So, if the data was written into the root directory, then all JavaFX programs on www.jfxia.com could read or write it.

By using a full path prefix, different JavaFX programs can choose to share or hide their data files using the `Storage` mechanism. Subdirectories (below home) help us to organize our data; parent directories (above home) help us to share it. It's *that* simple!

7.2.3 Adding pie and bar charts

For every Dilbert, there's a Pointy-Haired Boss, or so it seems. As programmers interested in graphics, we naturally want to spend our time writing video effects, UI controls, and Pac-Man clones; unfortunately, our bosses are unlikely to let us. When bosses say "graphics" they generally mean charts and graphs, and that's probably why the 1.2 release of JavaFX came with a collection of ready-made pie, bar, line, and other chart controls (like the 3D pie chart in figure 7.7).

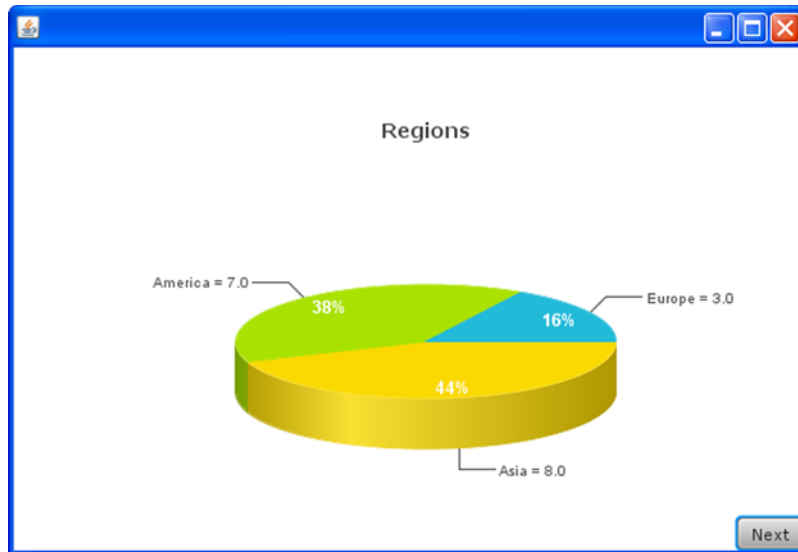


Figure 7.7 JavaFX has a powerful library of chart controls, including a 3D pie chart.

The next update of the feedback application will take the region and rating data collected from each user (and stored using the new persistent `Record` class) and produce a pie chart and a bar graph from it. The first part of the new `Feedback` class is shown in listing 7.7. The code continues in listings 7.8 and 7.9.

Listing 7.7 Feedback.fx (version 2, part 1)

```
package jfxia.chapter7;

import javafx.scene.Group;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.chart.BarChart;
import javafx.scene.chart.BarChart3D;
import javafx.scene.chart.PieChart;
import javafx.scene.chart.PieChart3D;
import javafx.scene.chart.part.CategoryAxis;
import javafx.scene.chart.part.NumberAxis;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.RadioButton;
import javafx.scene.control.Slider;
import javafx.scene.control.TextBox;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.Panel;
import javafx.scene.layout.Tile;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Alert;
```

```

import javafx.stage.Stage;
import javafx.util.Sequences;

var recordStore = Record.load();
def record:Record = Record {};
insert record into recordStore;

def winW:Number = 550;
def winH:Number = 350;
var showCharts:Boolean = false;
var showPie:Boolean = false;
def nextButton = Button {
  text: "Next";
  action: function() {
    if(not showCharts) {
      if(record.valid) {
        delete mainPan.content[0];
        Record.save(recordStore);
        insert createChartUI()
          before mainPan.content[0];
        showCharts=true;
      }
      else {
        Alert.confirm("Error", "Form incomplete.");
      }
    }
    else {
      showPie = not showPie;
    }
  }
}

```

Show form or charts?

Show pie or bar?

If form showing

Remove form from scene graph

Save form data

Insert charts into scene graph

Form invalid

Toggle between charts

// Part 2 is listing 7.8; part 3, listing 7.9

This listing focuses on the introduction of the record store and the changes to the Next button. The recordStore variable is a sequence of Record objects, one of each feedback entry recorded in the application's persistent storage. Having loaded past records, we append the virgin record created as a model for our feedback UI.

After the user completes the feedback form, clicking the Next button will record the data and take the user to the charts. Subsequent clicks of the Next button toggle between the Regions pie chart and the Ratings bar chart. Therefore, the nextButton control needs to be aware of whether it's moving from the form to the charts or toggling between the two charts. The showCharts and showPie variables are used for that very purpose.

Within the nextButton action handler the code first checks, courtesy of showCharts, which mode the UI is currently in (form or charts). If the form is currently displayed, the form UI is deleted from the scene graph, the Record sequence is saved to persistent storage (including the Record just populated by the form), and the chart UI is created and added to the scene graph. Finally showCharts is set to mark the switch from form to charts. If nextButton is clicked when showCharts is already set, the showPie variable is toggled, causing a different chart to be displayed.

Listing 7.7 contains all the necessary updates to the application, except the crucial step of creating the charts themselves. To find out how that's done, we need to consult listing 7.8, which holds the first part of the `createChartUI()` function. Because large parts of the class are unchanged, I've replaced the details with comments in bold.

Listing 7.8 Feedback.fx (version 2, part 2)

```
// Part 1 is listing 7.7
var mainPan:Panel;
Stage {
    // Stage details unchanged from previous
    // version
}

function createFeedbackUI() : Node {
    // createFeedbackUI() details unchanged from
    // previous version
}

function createRow(lab:String,n:Node,v:Node) : Node {
    // createRow() details unchanged from previous
    // version
}

function createChartUI() : Node {
    def regionData:PieChart.Data[] =
        for(r in Record.REGIONS) {
            var regIdx:Integer = indexof r;
            var cnt:Integer = 0;
            for(rc in recordStore) {
                if(rc.region == regIdx) cnt++;
            }

            PieChart.Data {
                label: r;
                value: cnt;
            }
        }

    var highestCnt:Integer = 0;
    def ratingData:BarChart.Data[] =
        for(i in [1..10]) {
            var cnt:Integer = 0;
            for(rec in recordStore) {
                def rat = rec.rating as Integer;
                if(rat == i) cnt++;
            }
            if(cnt>highestCnt) highestCnt=cnt;

            BarChart.Data {
                category: "{i}";
                value: cnt;
            }
        }
}
```

Refer to previous
Feedback.fx
version

Count respondents
for given region

Create PieChart.Data
for region

Highest rating
in data

Count respondents
for given rating

Create BarChart.Data
for rating

This is only the first half of the function, not the entire thing. It deals with creating the data required by the pie and bar chart controls. The first block of code builds the data for the regions pie chart, the second block deals with the ratings bar chart.

```
def regionData:PieChart.Data[] =
  for(r in Record.REGIONS) {
    var regIdx:Integer = indexOf r;
    var cnt:Integer = 0;
    for(rc in recordStore) {
      if(rc.region == regIdx) cnt++;
    }
    PieChart.Data {
      label: r;
      value: cnt;
    }
  }
```

The first block, reprised here, creates a `PieChart.Data` object for each region named in the `Record.REGIONS` sequence. These objects collect to create a sequence called `regionData`. For each region we walk over every record in `recordStore`, building a count of the responses for that region—the `regIdx` variable holds the region index, and the inner `for` loop looks for records matching that index. Once we have the count, we use it with the region name to create a new `PieChart.Data` object, which gets added to the `regionData` sequence.

```
var highestCnt:Integer = 0;
def ratingData:BarChart.Data[] =
  for(i in [1..10]) {
    var cnt:Integer = 0;
    for(rec in recordStore) {
      def rat = rec.rating as Integer;
      if(rat == i) cnt++;
    }
    if(cnt>highestCnt) highestCnt=cnt;
    BarChart.Data {
      category: "{i}";
      value: cnt;
    }
  }
```

The bar chart code, reproduced here, is very similar to the pie chart code. This time we're dealing with numeric ratings between 1 and 10; for each of the 10 possible ratings we scan the records, counting how many feedback responses gave that rating. We keep track of the size of the most popular (`highestCnt`) so we can scale the chart appropriately. Each rating's count is used to create a `BarChart.Data` object, used to populate the `ratingData` sequence.

Now that we've seen how the data in the record store is translated into sequences of `PieChart.Data` and `BarChart.Data` objects, we can turn our attention to the actual creation of the chart controls themselves. Listing 7.9 shows how it's done.

Listing 7.9 Feedback.fx (version 2, part 3)

```

Group {
  content: [
    PieChart3D {
      width: 500; height: 350;
      visible: bind showPie;
      title: "Regions";
      titleGap: 0;
      data: regionData;
      pieLabelVisible: true;
      pieValueVisible: true;
    },
    BarChart3D {
      width: 500; height: 350;
      visible: bind not showPie;
      title: "Ratings";
      titleGap: 10;
      data: BarChart.Series {
        name: "Ratings";
        data: ratingData;
      }
      categoryAxis: CategoryAxis {
        categories: for(r in ratingData)
          r.category;
      }
      valueAxis: NumberAxis {
        lowerBound: 0;
        upperBound: highestCnt;
        tickUnit: 1;
      }
    }
  ]
}

```

Pie chart control

Control its visibility

Plug in data sequence

Bar chart control

Control its visibility

Wrap data sequence

Labels for category axis

Bounds and unit for value axis

The final part of the `createChartUI()` function manufactures the chart scene graph. Both chart controls are held inside a simple `Group`. The first, `PieChart3D`, has its visibility bound to show whenever `showPie` is true; the second, `BarChart3D`, has its visibility bound to show whenever `showPie` is false. Figure 7.8 reveals how both charts look when displayed.

The `PieChart3D` control's declaration is quite simple to understand: the title and `titleGap` properties control the label that appears over the chart and how far away it is from the bounds of the pie graphic itself. The `regionData` sequence we created in listing 7.8 is referenced via `data`. The `pieLabelVisible` and `pieValueVisible` variables cause the values and labels of each pie slice (set in each `PieChart.Data` object, as you recall) to be drawn.

The `BarChart3D` control's declaration is more complicated. As well as the familiar title and `titleGap` properties, objects controlling each axis are created, and the `ratingData` sequence (created in listing 7.8) is wrapped inside a `BarChart.Series`

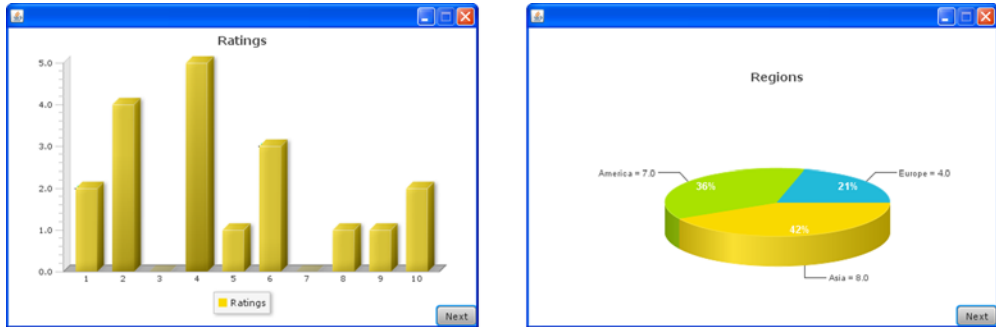


Figure 7.8 A bar chart and a pie chart, as drawn in 3D by the JavaFX 1.2 chart library

object rather than being plugged directly into the control. But what is a `BarChart.Series`, and why is it necessary?

JavaFX's bar charts are quite clever beasts—they can display several sets of data simultaneously. Suppose our application had allowed the user to set two ratings instead of just one; we could display both sets (or *series*) in one chart, looking like figure 7.9.

Each category in the chart (1 to 10 along the horizontal axis) has two independent bars, one from each series of data.

```
data: [
    BarChart.Series {
        name: "Ratings";
        data: ratingData;
    },
    BarChart.Series {
        name: "Ratings 2";
        data: reverse ratingData;
    }
]
```

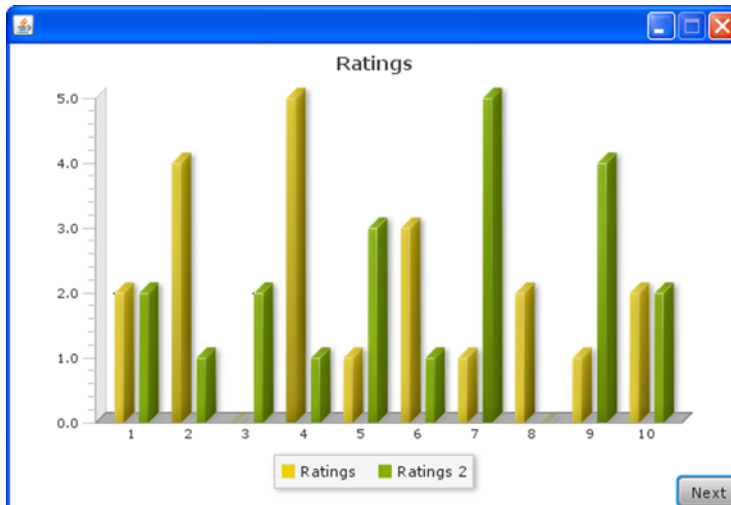


Figure 7.9 This is what the project's bar chart would look like if it used two data series rather than one. Each category has two bars, and the key at the foot of the chart shows two labels.

The data property in `BarChart3D` accepts a sequence of type `BarChart.Series` rather than just a single object. The example quick-'n'-dirty code snippet proves this, by adding a second series that mirrors the first (this was how figure 7.9 was created). Each `BarChart.Series` object contains a `BarChart.Data` object holding its data, and a label to use in the key at the foot of the chart.

With that explained, only the `categoryAxis` and `valueAxis` variables remain a mystery. We'll look at those in the next section.

7.2.4 Taking control of chart axes

If you've checked out the JavaFX API documentation, you'll have noticed that the chart classes are grouped into three packages. The main one, `javafx.scene.chart`, contains the chart controls themselves, plus their associated data and series classes. The `javafx.scene.chart.data` package holds the superclasses for these data and series classes (unless you code your own charts, you're unlikely to work directly with them). `javafx.scene.chart.part` contains classes to model common chart elements, such as the axes. This package is home to the `CategoryAxis` and `NumberAxis` used in listing 7.9. Here's a reminder of that code:

```
categoryAxis: CategoryAxis {
    categories: for(r in ratingData)
        r.category;
}
valueAxis: NumberAxis {
    lowerBound: 0;
    upperBound: highestCnt;
    tickUnit: 1;
}
```

These variables (part of the `BarChart3D` control, as you recall) determine how the horizontal (category) and vertical (value) axes will be drawn. Although many different types of charts exist, there are only a handful of different axis *flavors*. An axis can represent a linear (analog) range of values, or it can represent a finite set of groups. For example, if we drew a line chart of temperature readings over a 24-hour period, it might have *ticks* (or labels) every 5 degrees Centigrade on the temperature axis and every hour on the time axis, but the data is not necessarily constrained by these markings. We might plot a temperature of 37 degrees (which is not a multiple of 5) at 9:43 a.m. (which is not on the hour). If we then drew a bar chart of average daily temperatures for each month in a year, while the temperature axis would still be analog, the time axis would now be grouped into 12 distinct categories (January, February, etc.).

`CategoryAxis` and `NumberAxis` (plus the abstract `ValueAxis`) model those axis types. The first handles grouped data and the second linear data. Each has a host of configuration options for controlling what gets drawn and how. One particularly useful example is `NumberAxis.formatTickLabel()`, a function type that allows us to control how the values on an axis are labeled. To convert the values 0 to 11 into calendar names, we would use the following `NumberAxis` code:

```

NumberAxis {
    label: "Months";
    lowerBound:0;  upperBound:12;
    tickUnit: 1;
    formatTickLabel: function(f:Float) {
        def cal = ["Jan", "Feb", "Mar", "Apr", "May", "Jun" ,
                  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" ];
        cal[f as Integer];
    }
}

```

The `lowerBound` and `upperBound` variables constrain the axis to values between 0 to 11 (inclusive), while `tickUnit` determines the frequency of labels on the axis (one label for every integer value). The `formatTickLabel` code converts a label value into text, using a lookup table of calendar month names. (The `cal` sequence is a local variable merely for the sake of brevity; in real-world code it would be declared at the script or class level, thereby avoiding a rebuild every time the function runs.)

You can see more examples of these axis classes at work in the next section, where we look at the other types of chart supported by JavaFX.

7.2.5 Other chart controls (area, bubble, line, and scatter)

Bar charts and pie charts are staples of the charting world, but JavaFX offers far more than just these two. In this section we'll tour alternatives, beginning with the two charts in figure 7.10.

The `AreaChart` class accepts a collection of x and y values, using them as points defining a polygon. The effect looks like a mountain range, and it's a useful alternative to the simple line chart (see later).

As well as the 3D bar chart we witnessed in the feedback project, JavaFX has a 2D alternative. The `BarChart` control has the same data requirements as its `BarChart3D` sibling; both accept a collection of values defining the height of each bar.

Two more chart types are demonstrated in figure 7.11.

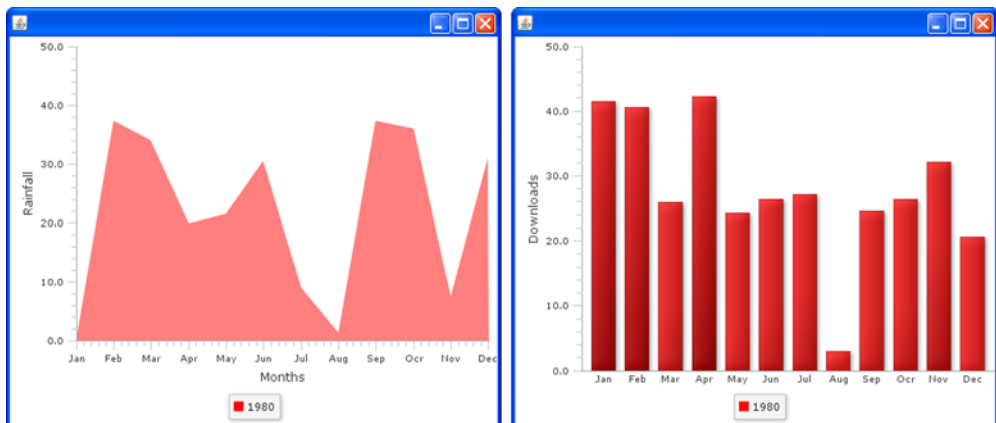


Figure 7.10 An area chart (left) and a standard 2D bar chart (right)

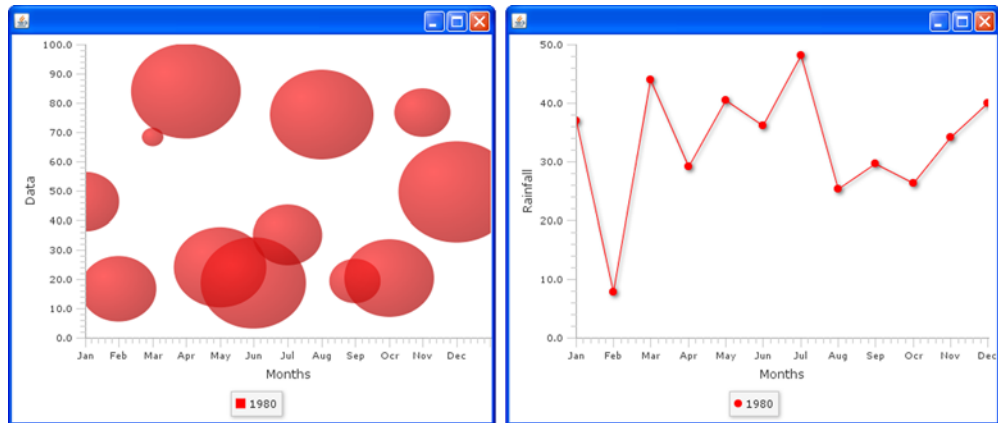


Figure 7.11 A bubble chart (left) and a line chart (right)

The BubbleChart is rarely used, but it can be a real lifesaver when necessary. As well as x and y values, each point can contain a radius, giving effectively a third dimension of data.

The LineChart is the familiar graph we all remember from school. Using x and y values, it plots a line to connect the dots.

The two final chart types are shown in figure 7.12.

Just as the bar chart comes in 2D and 3D flavors, so the pie chart also has 2D and 3D variations. Both PieChart and PieChart3D are unusual, as (understandably) they do not support multiple series of data like the other charts; only one set of values can be plugged in.

The ScatterChart, although graphically the simplest of all JavaFX charts, is incredibly useful for revealing patterns within a given body of data. It plots x and y values anywhere within its coordinate space.

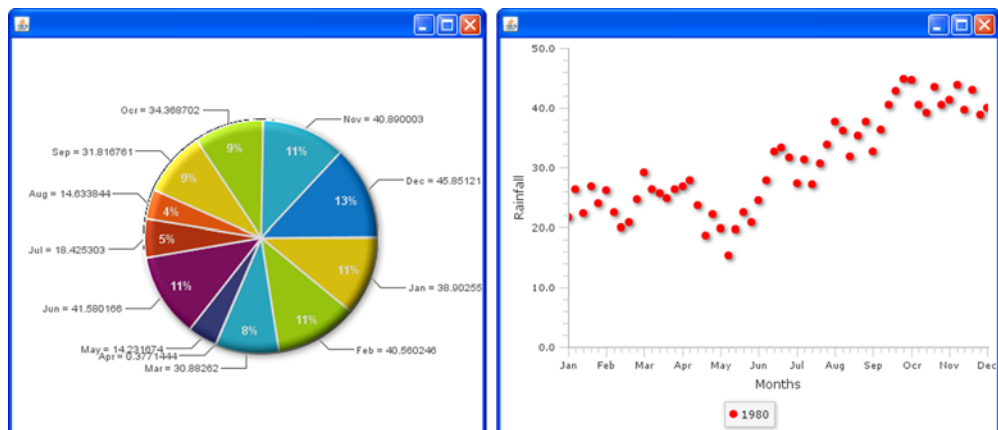


Figure 7.12 A standard 2D pie chart (left) and a scatter chart (right)

That completes our whistle-stop tour of the JavaFX charting controls. All that remains is to compile and try our new feedback application.

7.2.6 Running version 2

Starting up the project gives us the same form UI from version 1, but now the Next button has been rewired to save the data and replace the form UI with a couple of charts. The result should look like figure 7.13. Depending on how powerful your computer is, the flip between form and charts may not be instantaneous. I ran the code on an old laptop, and there was a slight (0.5s to 1s) delay before the charts appeared. A brief investigation revealed the delay emanated from building the chart UI. (Perhaps a simple *processing* animation is needed, giving an instance response to a Next button click?)

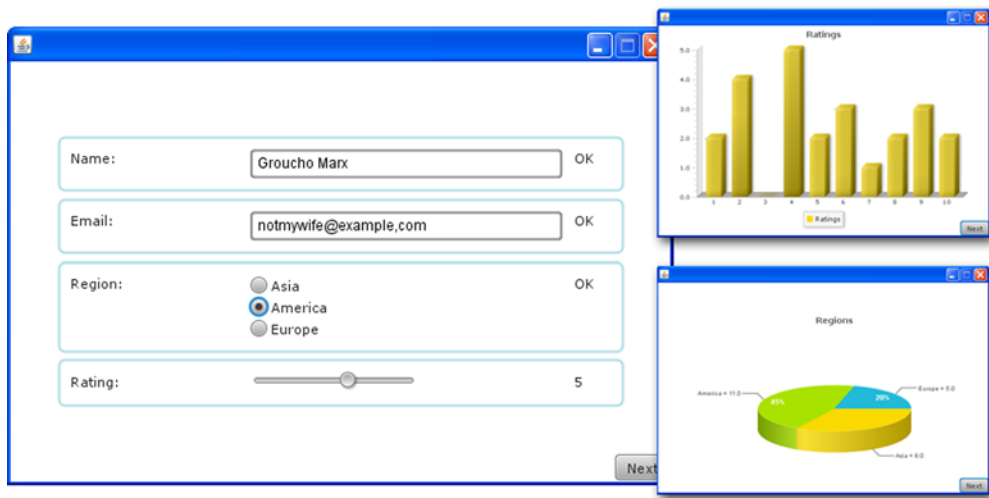


Figure 7.13 Version 2 of the Feedback application runs, complete with form (main image) and two charts (thumbnails).

Once the form has been submitted, to enter another record you need to quit and restart the application. I could have added logic to reset the UI and add a new Record to the recordStore, but the listings would have been longer, with little extra educational value. You can add the code yourself if you want. (Hint: you need to make sure the UI is bound against a new Record object; either rebuild a fresh UI for each Record or devise an abstraction, so the UI isn't plumbed directly into a specific Record.)

In the final part of this chapter, we finish with a super-sized bonus section, all about creating our own controls and skinning them.

7.3 Bonus: creating a styled UI control in JavaFX

In this chapter's project we saw JavaFX controls in action; now it's time to write our own control from scratch. Because much of the detail in this section is based on research and trial and error rather than official JavaFX documentation, I've separated

it from the main project in this chapter and turned it into a (admittedly, rather lengthy) bonus project.

As already explained, JavaFX's controls library, in stark contrast to Java's Swing library, can function across a wide range of devices and environments—but that's not all it can do. Each control in the library can have its appearance customized through skins and stylesheets. This allows designers and other nonprogrammers to radically change the look of an application, without having to write or recompile any code.

In this section we'll develop our own very simple, style-aware control. Probably the simplest of all widget types is the humble progress bar; it's a 100% visual element with no interaction with the mouse or keyboard, and as such it's ideal for practicing skinning. The standard controls API already has a progress control of its own (we'll see it in action in the next chapter), but it won't hurt to develop our own super-sexy version.

WARNING *Check for updates* This section was written originally against JavaFX 1.1 and then updated for the 1.2 release. Although the latter saw the debut of the controls library, at the time of writing much of the details on how skins and Cascading Style Sheets (CSS) support works is still largely undocumented. The material in this section is largely based on what little information there is, plus lots of investigation. Sources close to the JavaFX team have hinted that, broadly speaking, the techniques laid out in the following pages are correct. However, readers are urged to search out fresh information that may have emerged since this chapter was written, particularly official tutorials, or best practice guides for writing skins.

Let's look at stylesheets.

7.3.1 What is a stylesheet?

Back in the old days (when the web was in black and white) elements forming an HTML document determined both the logical meaning of their content and how they would be displayed. For example, a `<p>` element indicated a given node in the DOM (Document Object Model) was of paragraph type. But marking the node as a paragraph also implied how it would be drawn on the browser page, how much whitespace would appear around it, how the text would flow, and so on. To counteract these presumptions new element types like `` and `<center>` were added to browsers to influence display. With no logical function in the document, these new elements polluted the DOM and made it impossible to render the document in different ways across different environments.

CSS is a means of fixing this problem, by separating the logical meaning of an element from the way it's displayed. A *stylesheet* is a separate document (or a self-contained section within an HTML document) defining *rules* for rendering the HTML elements. By merely changing the CSS, a web designer can specify the display settings of a paragraph (for example), without need to inject extra style-specific elements into the HTML. Stylesheet rules can be targeted at every node of a given type, at nodes having been assigned a given class, or at a specific node carrying a given ID

(see figure 7.14). Untangling the logical structure of a document from how it should be displayed allows the document to be shown in many different ways, simply by changing its stylesheet.

What works for web page content could also work for GUI controls; if buttons, sliders, scrollbars, and other controls deferred to an external stylesheet for their display settings, artists and designers could change their look without having to write a single line of software code. This is precisely the intent behind JavaFX's style-aware controls library.

JavaFX allows both programmers and designers to get in on the style act. Each JavaFX control defers not to a stylesheet directly but to a skin. Figure 7.15 shows this relationship diagrammatically. The skin is a JavaFX class that draws the control; it can expose various properties (public instance variables), which JavaFX can then allow designers to change via a CSS-like file format.

In a rough sense the control acts as the *model* and the skin as the *view*, in the popular Model/View/Controller model of UIs. But the added JavaFX twist is that the skin can be configured by a stylesheet. Controls are created by subclassing `javafx.scene.control.Control` and skins by subclassing `Skin` in the same package. Another class, `Behavior`, is designed to map inputs (keystrokes, for example) to host-specific actions on the control, effectively making it a type of MVC *controller*.

Now that you understand the theory, let's look at each part in turn as actual code.

7.3.2 Creating a control: the *Progress* class

We're going to write our own style-compliant control from scratch, just to see how easy it is. The control we'll write will be a simple progress bar, like the one that might appear during a file download operation. The progress bar will take minimum and maximum bounds, plus a value, and display a series of boxes forming a horizontal bar, colored to show where the value falls in relation to its bounds. To keep things nice and simple our control won't respond to any mouse or keyboard input, allowing us to focus exclusively on the interaction between model (control) and view (skin), while ignoring the controller (behavior).

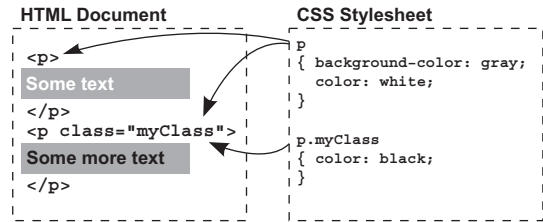


Figure 7.14 Two style rules and two HTML paragraph elements. The first rule applies to both paragraphs, while the second applies only to paragraphs of class `myClass`.

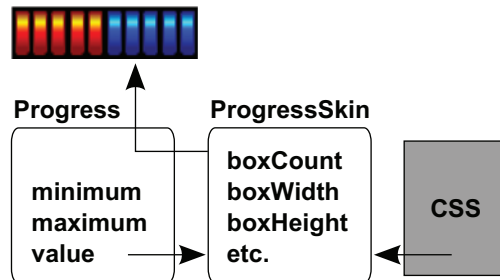


Figure 7.15 The data from the control (*Progress*) and the CSS from the stylesheet are combined inside the skin (*ProgressSkin*) to produce a displayable UI control, in this example, a progress bar.

Listing 7.10 shows the control class itself. This is the object other software will use when wishing to create a control declaratively. It subclasses the `Control` class from `javafx.scene.control`, a type of `CustomNode` designed to work with styling.

Listing 7.10 `Progress.fx`

```
package jfxia.chapter7;

import javafx.scene.control.Control;

public class Progress extends Control
{
    public var minimum:Number = 0;
    public var maximum:Number = 100;
    public var value:Number = 50 on replace {
        if (value < minimum) { value = minimum; }
        if (value > maximum) { value = maximum; }
    };

    override var skin = ProgressSkin{}; ← Override the skin
}
```

**The control's
data**

Our `Progress` class has three variables: `maximum` and `minimum` determine the range of the progress (its high and low values), while `value` is the current setting within that range. We override the `skin` variable inherited from `Control` to assign a default skin object. The skin, as you recall, gives our control its face and processes user input. It's a key part of the styling process, so let's look at that class next.

7.3.3 Creating a skin: the `ProgressSkin` class

In itself the `Progress` class does nothing but hold the fundamental data of the progress meter control. Even though it's a `CustomNode` subclass, it defers all its display and input to the skin class. So, what does this skin class look like? It looks like listing 7.11.

Listing 7.11 `ProgressSkin.fx (part 1)`

```
package jfxia.chapter7;

import javafx.scene.Group;
import javafx.scene.control.Skin;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Paint;
import javafx.scene.paint.Stop;
import javafx.scene.shape.Rectangle;

public class ProgressSkin extends Skin {
    public var boxCount:Integer = 10;
    public var boxWidth:Number = 7;
    public var boxHeight:Number = 20;
    public var boxStrokeWidth:Number = 1;
    public var boxCornerArc:Number = 3;
    public var boxHGap:Number = 1;

    public var boxUnsetStroke:Paint = Color.DARKBLUE;
```

**These properties
can be styled
with CSS**

```

public var boxUnsetFill:Paint = makeLG(Color.CYAN,
    Color.BLUE,Color.DARKBLUE);
public var boxSetStroke:Paint = Color.DARKGREEN;
public var boxSetFill:Paint = makeLG(Color.YELLOW,
    Color.LIMEGREEN,Color.DARKGREEN);

def boxValue:Integer = bind {
    var p:Progress = control as Progress;
    var v:Number = (p.value-p.minimum) /
        (p.maximum-p.minimum);
    (boxCount*v) as Integer;
}
// ** Part 2 is listing 7.12

```

These properties can be styled with CSS

How many boxes to highlight?

Listing 7.11 is the opening of our skin class, `ProcessSkin`. (The second part of the source code is shown in listing 7.12.) The `Progress` class is effectively the model, and this class is the view in the classic MVC scheme. It subclasses `javafx.scene.control.Skin`, allowing it to be used as a skin.

The properties at the top of the class are all exposed so that they can be altered by a stylesheet. They perform various stylistic functions.

- The `boxCount` variable determines how many progress bar boxes should appear on screen.
- The `boxWidth` and `boxHeight` variables hold the dimensions of each box, while `boxHGap` is the pixel gap between boxes.
- The variable `boxStrokeWidth` is the size of the trim around each box, and `boxCornerArc` is the radius of the rounded corners.
- For the public interface, we have two pairs of variables that colorize the control. The first pair are `boxUnsetStroke` and `boxUnsetFill`, the trim and body colors for *switched-off* boxes; the second pair is (unsurprisingly) `boxSetStroke` and `boxSetFill`, and they do the same thing for *switched-on* boxes. The `makeLG()` function is a convenience for creating gradient fills; we'll see it in the concluding part of the code.
- The private variable `boxValue` uses the data in the `Progress` control to work out how many boxes should be switched on. The reference to `control` is a variable inherited from its parent class, `Skin`, allowing us to read the current state of the control (model) the skin is plugged into.

One thing of particular note in listing 7.11: the stroke and fill properties are `Paint` objects, not `Color` objects. Why? Quite simply, the former allows us to plug in a gradient fill or some other complex pattern, while the latter would support only a flat color. And, believe it or not, JavaFX's support for styles actually extends all the way to patterned fills.

Moving on, the concluding part of the source code (listing 7.12) shows how these variables are used to construct the progress meter.

Listing 7.12 `ProgressSkin.fx` (part 2)

```

// ** Part 1 is listing 7.11
    override var node = HBox {
        spacing: bind boxHGap;

```

Override node in Skin class

```

content: bind for(i in [0..

Bind visible properties



Bind trim color to boxValue



Bind body color to boxValue



Inherited from Resizable



Gradient paint from three colors


```

We see how the style variables are used to form a horizontal row of boxes. An inherited variable from Skin, named `node`, is used to record the skin's scene graph. Anything plugged into `node` becomes the corporeal form (physical body) of the control the skin is applied to. In our case we've a heavily bound sequence of `Rectangle`

objects, each tied to the instance variables of the class. This sequence is all it takes to create our progress bar.

Because our control needs to be capable of being laid out, our `Skin` subclass overrides functions to expose its maximum, minimum, and preferred dimensions. To keep the code small I've used the preferred size for all three, and I've ignored the available width/height passed in as a parameter (which is a bit naughty). Our skin also fills out a couple of abstract functions, `contains()` and `intersects()`, by deferring to the control. (Simply redirecting these calls to the control like this should work for the majority of custom controls you'll ever find yourself writing.) Remember, even though the control delegates its appearance to the skin, it is still a genuine scene graph node, and we can rely on its inherited functionality.

At the end of the listing is the `makeLG()` function, a convenience for creating the `LinearGradient` paints used as default values for the box fills.

All that remains, now that we've seen the control and its skin, is to take a look at it running with an actual style document.

7.3.4 Using our styled control with a CSS document

The `Progress` and `ProgressSkin` classes create a new type of control, capable of being configured through an external stylesheet document. Now it's time to see how our new control can be used and manipulated.

Listing 7.13 is a test program for trying out our new control. It creates three examples: (1) a regular `Progress` control without any *class* or *ID* (note, in this context the word *class* refers to the CSS-style class and has nothing to do with any class written in the JavaFX Script language), (2) another `Progress` control with an ID ("testID"), and (3) a final `Progress` assigned to a style class ("testClass").

Listing 7.13 TestCSS.fx

```
package jfxia.chapter7;

import javafx.animation.KeyFrame;
import javafx.animation.Interpolator;
import javafx.animation.Timeline;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;

var val:Number = 0;
Stage {
    scene: Scene {
        content: VBox {
            spacing:10;
            layoutX: 5; layoutY: 5;
            content: [
                Progress {
                    minimum: 0; maximum: 100;
                    value: bind val;
                },
                Progress {
                    ID: "testID";
                    styleClass: "testClass";
                }
            ]
        }
    }
}
```

Plain control,
no ID or class

Control with ID


```

        id: "testId";
        minimum: 0; maximum: 100;
        value: bind val;
    },
    Progress {
        styleClass: "testClass";
        style: "boxSetStroke: white";
        minimum: 0; maximum: 100;
        value: bind val;
    }
];

};
stylesheets: [ "{__DIR__}Test.css" ]
fill: Color.BLACK;
width: 230; height: 105;
};
title: "CSS Test";
};

Timeline {
    repeatCount: Timeline.INDEFINITE;
    autoReverse: true;
    keyFrames: [
        at(0s) { val => 0 } ,
        at(0.1s) { val => 0 tween Interpolator.LINEAR } ,
        at(0.9s) { val => 100 tween Interpolator.LINEAR } ,
        at(1s) { val => 100 }
    ];
}.play();

```

Control
with class

Assign
stylesheets

Run val backward
and forward

Note how the final progress bar also assigns its own local style for the `boxSetStroke`? This is important, as we'll see in a short while.

Figure 7.16 shows the progress control on screen. All three Progress bars are bound to the variable `val`, which the Timeline at the foot of the code repeatedly increases and decreases (with a slight pause at either end), to make the bars shoot up and down from minimum to maximum.

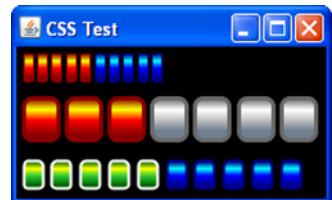


Figure 7.16 Three examples of our progress bar in action

The key part of the code lies in the `stylesheets` property of `Scene`. This is where we plug in our list of CSS documents (just one in our example) using their URLs. This particular example expects our docs to sit next to the `TestCSS` bytecode files. The `__DIR__` built-in variable returns the directory of the current class file as a URL, as you recall. If you downloaded the project's source code, you'll find the CSS file nested inside the `res` directory, off the project's root. When you build the project, make sure this file gets copied into the build directory, next to the `TestCSS` class file.

Now it's time for the grand unveiling of the CSS file that styles our component. Listing 7.14 shows the three style rules we've created for our test program. (Remember, it lives inside `jfxia.chapter7`, next to the `TestCSS` class.)

Classes and IDs

In stylesheets, classes and IDs perform similar functions but for use in different ways. Assigning an ID to an element in a document means it can be targeted specifically. IDs are supposed to be unique names, appearing only once in a given document. They can be used for more than just targeting stylesheet rules.

What happens if we need to apply a style, not to a specific DOM element but to an entire subset of elements? To do this we use a class, a nonunique identifier designed primarily as a type identifier onto which CSS rules can bind.

Listing 7.14 Test.css

```
"jfxia.chapter7.Progress" {
    boxSetStroke: darkred;
    boxSetFill: linear (0%,0%) to (0%,100%) stops
        (0.0,darkred), (0.25,yellow), (0.50,red), (0.85,darkred);
    boxUnsetStroke: darkblue;
    boxUnsetFill: linear (0%,0%) to (0%,100%) stops
        (0.0,darkblue), (0.25,cyan), (0.50,blue), (0.85,darkblue);
}

"jfxia.chapter7.Progress"#testId {
    boxWidth: 25; boxHeight: 30;
    boxCornerArc: 12; boxStrokeWidth: 3;
    boxCount: 7;
    boxHGap: 1;

    boxUnsetStroke: dimgray;
    boxUnsetFill: linear (0%,0%) to (0%,100%) stops
        (0%,dimgray), (25%,white), (50%,silver), (75%,slategray);
}

"Progress".testClass {
    boxWidth: 14;
    boxCornerArc: 7;
    boxStrokeWidth: 2;
    boxHGap: 3;

    boxSetStroke: darkgreen;
    boxSetFill: linear (0%,0%) to (0%,100%) stops
        (0.0,darkgreen), (0.25,yellow), (0.50,limegreen), (0.85,darkgreen);
}
```

The first rule targets all instances of `jfxia.chapter7.Progress` controls. The settings in its body are applied to the skin of the control. The second is far more specific; like the first it applies to `jfxia.chapter7.Progress` controls, but only to that particular control with the ID of `testID`. The final rule targets the `Progress` class again (this time omitting the package prefix, just to show it's not necessary if the class name alone is not ambiguous), but it applies itself to any such control belonging to the style class `testClass`.

If multiple rules match a single control, the styles from all rules are applied in a strict order, starting with the generic (no class, no ID) rule, continuing with the class rule, then the specific ID rule, and finally any style plugged directly into the object literal inside the JavaFX Script code. Remember that explicit style assignment I said was important a couple of pages back? That was an example of overruling the CSS file with a style written directly into the JFX code itself. The styles for each matching rule are applied in order, with later styles overwriting the assignments of previous styles.

If absolutely nothing matches the control, the default styles defined in the skin class itself, `ProgressSkin` in our case, remain untouched. It's important, therefore, to ensure your skins always have sensible defaults.

You'll note how the class name is wrapped in quotes. If you were wondering, this is simply to stop the dots in the name from being misinterpreted as CSS-style class separators, like the one immediately before the name `"testClass"`.

Cascading Style Sheets

In this book we don't have the space to go into detail about the format of CSS, on which JavaFX stylesheets are firmly based. CSS is a World Wide Web Consortium specification, and the W3C website has plenty of documentation on the format at <http://www.w3.org/Style/CSS/>

Inside the body of each style rule we see the skin's public properties being assigned. The majority of these assignments are self-explanatory. Variables like `boxCount`, `boxWidth`, and `boxHeight` all take integer numbers, and color variables can take CSS color definitions or names, but what about the strange linear syntax?

7.3.5 Further CSS details

The exact nature of how CSS interacts with JavaFX skins is still not documented as this chapter is being written and updated, yet already several JFX devotees have dug deep into the class files and discovered some of the secrets therein.

In lieu of official tutorials and documentation, we'll look at a couple of examples to get an idea of what's available. An internet search will no doubt reveal further styling options, although by the time you read this the official documentation should be available.

Here is one of the linear paint examples from the featured stylesheet:

```
boxUnsetFill: linear (0%,0%) to (0%,100%) stops  
    (0.0,dimgray), (0.25,white), (0.50,silver), (0.75,slategray);
```

The example creates, as you might expect, a `LinearGradient` paint starting in the top-left corner (0% of the way across the area, 0% of the way down) and ending in the bottom left (0% of the way across, 100% of the way down). This results in a straightforward vertical gradient. To define the color stops, we use a comma-separated list of

position/color pairs in parentheses. For the positions we could use percentages again or a fraction-based scale from 0 to 1. The colors are regular CSS color definitions (see the W3C's documentation for details).

The stylesheet in our example is tied directly to the `ProgressSkin` we created for our `Progress` control. The settings it changes are the publicly accessible variables inside the skin class. But we can go further than only tweaking the skin's variables; we can replace the entire skin class:

```
"Progress"#testID {  
    skin: jfxia.chapter7.AlternativeProgressSkin;  
}
```

The fragment of stylesheet in the example sets the skin itself, providing an alternative class to the `ProgressSkin` we installed by default. Obviously we haven't written such a class—this is just a demonstration. The style rule targets a specific example of the `Progress` control, with the ID `testID`, although if we removed the ID specifier from the rule it would target all `Progress` controls.

STYLING BUG

The JavaFX 1.2 release introduced a bug: controls created after the stylesheet was installed are not styled. This means if your application dynamically creates bits of UI as it runs and adds them into the scene graph, styling will not be applied to those controls. A bug fix is on its way; in the meantime the only solution is to remove and reassign the stylesheet's variable in `Scene` every time you create a new control that needs styling.

The `AlternativeProgressSkin` class would have its own public interface, with its own variables that could be styled. For this reason the rule should be placed before any other rule that styles variables of the `AlternativeProgressSkin` (indeed some commentators have noted it works best when placed in a separate rule, all on its own).

7.4 Summary

In this chapter we built a simple user interface, using JavaFX's controls API, and displayed some statistics, thanks to the charts API. We also learned how to store data in a manner that won't break as our code travels from device to device. Although the project was simple, it gave a solid grounding into controls, charts, and client-side persistence. However, we didn't get a chance to look at every type of control.

So, what did we miss? `CheckBox` is a basic opt-in/out control, either checked or unchecked. JavaFX check boxes also support a third option, *undefined*, typically used in check box trees, when a parent check box acts as a master switch to enable/disable all its children. `Hyperlink` is a web-like link, acting like a `Button` but looking like a piece of text. `ListView` displays a vertical list of selectable items. `ProgressBar` is a long, thin control, showing either the completeness of a given process or an animation suggesting work is being done; `ProgressIndicator` does the same thing but with a more compact *dial* display. `ScrollBar` is designed to control a large area displayed

within a smaller viewport. `ToggleButton` flips between selected or unselected; it can be used in a `ToggleGroup`; however (unlike a `RadioButton`), a `ToggleButton` can be unselected with a second click, leaving the group with no currently selected button.

In the bonus project we created our own control that could be manipulated by CSS-like stylesheets. Although some of the styling detail was a little speculative because of the unavailability of solid documentation at the time of writing, the project should, at the very least, act as a primer.

With controls and charts we can build serious applications, targeted across a variety of platforms. With skins and styling they can also look good. And, since everything is scene graph-based, it can be manipulated just like the graphics in previous chapters.

In the next chapter we're sticking with the practical theme by looking at web services—but, of course, we'll also be having plenty of fun. Until then, why not try extending the main project's form with extra controls or charts? Experiment, see what works, and get some positive feedback.