

JavaFX

In this chapter

- 13.1 A Brief History of Java GUI Programming, page 1
- 13.2 Displaying Information in a Scene, page 3
- 13.3 Event Handling, page 16
- 13.4 Layout, page 28
- 13.5 User Interface Controls, page 47
- 13.6 Properties and Bindings, page 82
- 13.7 Long-Running Tasks in User Interface Callbacks, page 91

JavaFX is a user interface toolkit for writing rich client applications with Java. It is bundled with some versions of Java 7 through 10, and is available through the OpenJFX project (<https://wiki.openjdk.java.net/display/OpenJFX/Main>) for newer versions of Java. In this chapter, you will learn the basics of JavaFX development.

13.1 A Brief History of Java GUI Programming

When Java was born, the Internet was in its infancy and personal computers were on every desktop. Business applications were implemented with “fat clients”—programs with lots of buttons and sliders and text fields that communicated with a server. This was considered a lot nicer than the “dumb terminal” applications from an even earlier era. Java 1.0 included the AWT,

a toolkit for graphical user interfaces, that had the distinction of being cross-platform. The idea was to serve up the fat clients over the nascent Web, eliminating the cost of managing and updating the applications on every desktop.

The AWT had a noble idea: provide a common programming interface for the native buttons, sliders, text fields, and so on of various operating systems. But it didn't work very well. There were subtle differences in the functionality of the user interface controls in each operating system, and what should have been "write once, run anywhere" turned into "write many times, debug everywhere."

Next came Swing. The central idea behind Swing was not to use the native controls, but to paint its own. That way, the user interface would look and feel the same on every platform. Or, if users preferred, they could ask for the native look-and-feel of their platform, and the Swing controls would be painted to match the native ones. Of course, all that painting was slow, and users complained. After a while, computers got faster, and users complained that Swing was ugly—indeed, it had fallen behind the native controls that had been spruced up with animations and fancy effects. More ominously, Flash was increasingly used to create user interfaces with even flashier effects that didn't use the native controls at all.

In 2007, Sun Microsystems introduced a new technology, called JavaFX, as a competitor to Flash. It ran on the Java VM but had its own programming language, called JavaFX Script. The language was optimized for programming animations and fancy effects. Programmers complained about the need to learn a new language, and they stayed away in droves. In 2011, Oracle released a new version, JavaFX 2.0, that had a Java API and no longer needed a separate programming language. As of Java 7 update 6, JavaFX 2.2 has been bundled with the JDK and JRE. Since it wouldn't be a true part of Java if it didn't have crazy jumps in version numbers, the version accompanying Java 8 was called JavaFX 8. JavaFX versions 9 and 10 were bundled with Java 9 and 10.

Of course, Flash is now a bad memory, and most user interfaces live in a browser or a mobile device. Still, there are situations where a "fat client" on a desktop makes users more productive. Also, Java now runs on ARM processors, and there are embedded systems that need user interfaces, such as kiosks and in-car displays. Why didn't Oracle just put the good parts of JavaFX into Swing? Swing would have to be redesigned from the ground up to run efficiently on modern graphics hardware. Oracle decided that it wasn't worth the trouble. In fact, as of Java 11, Oracle doesn't even think it worth the trouble bundling JavaFX with Java. Hopefully, JavaFX will continue to thrive as an open source project.

In this chapter, we go over the basics of writing user interfaces in JavaFX, focusing on boring business applications with buttons, sliders, and text fields, not the flashy effects that were the original motivation behind JavaFX.

13.2 Displaying Information in a Scene

In the following sections, you will learn about the basic architecture of a JavaFX application. You will also see how to write simple JavaFX programs that display text and shapes.

13.2.1 Our First JavaFX Application

Let's start with a simple program that shows a message (see Figure 13.1). We use a *text node* to show the message and set the *x*- and *y*-position so that the message is approximately centered. The base point of the first character in the string will start at a position 75 pixels to the right and 100 pixels down. (You will see later in this chapter how to position text precisely.)

```
Text message = new Text(75, 100, "Not a Hello World program");
```



Figure 13.1 A window that displays information

Anything that you display in JavaFX is a *Node*. This includes both shapes and user interface controls. You collect nodes in a *Parent* (a *Node* that can organize other nodes) called the *root node*. If you don't need automatic positioning of nodes, use a *Pane* as the root.

It is also a good idea to set a preferred size for the pane. Otherwise, the pane is sized to exactly hold the shapes, without a margin.

```
Pane root = new Pane(message);  
root.setPrefSize(PREFERRED_WIDTH, PREFERRED_HEIGHT);
```

Then you construct a *scene* from the pane.

```
Scene scene = new Scene(root);
```

Next, the scene must reside in a *stage*, a window on a desktop (see Figure 13.2). The stage is passed as a parameter to the start method that you override in a subclass of the `Application` class. You can optionally set a window title. Finally, call the `show` method to show the window.

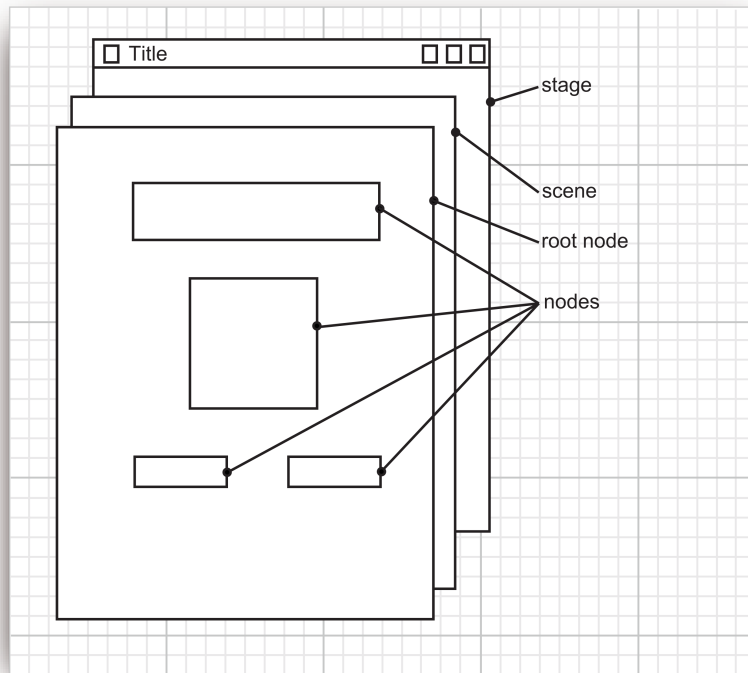


Figure 13.2 Internal structure of a stage

```
public class NotHelloWorld extends Application  
{  
    public void start(Stage stage)  
    {  
        . . .  
    }  
}
```

```

        stage.setScene(scene);
        stage.setTitle("NotHelloWorld");
        stage.show();
    }
}

```

You can see the complete program in Listing 13.1. The UML diagram in Figure 13.3 shows the relationships between the JavaFX classes that we use in this program.

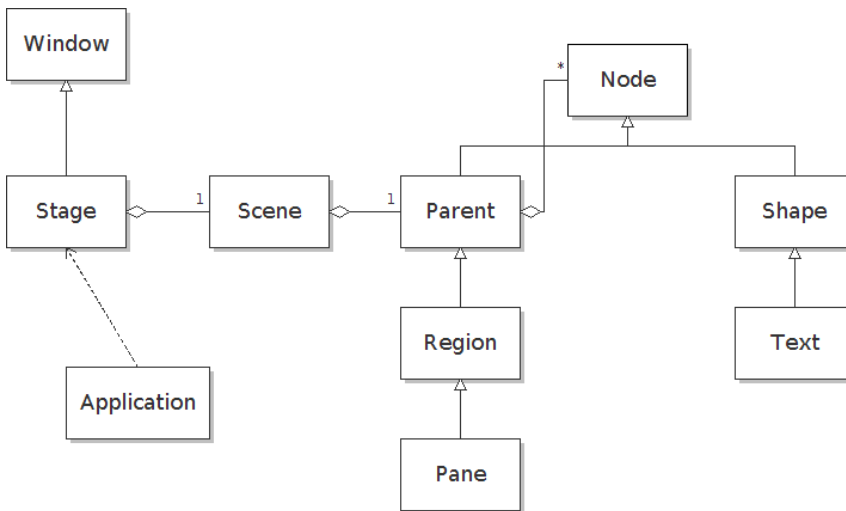


Figure 13.3 Relationships between core JavaFX classes

Listing 13.1 notHelloWorld/NotHelloWorld.java

```

1 package notHelloWorld;
2
3 import javafx.application.*;
4 import javafx.scene.*;
5 import javafx.scene.layout.*;
6 import javafx.scene.text.*;
7 import javafx.stage.*;
8
9 /**
10  @version 1.4 2017-12-23
11  @author Cay Horstmann
12 */

```

(Continues)

Listing 13.1 *(Continued)*

```
13 public class NotHelloWorld extends Application
14 {
15     private static final int MESSAGE_X = 75;
16     private static final int MESSAGE_Y = 100;
17
18     private static final int PREFERRED_WIDTH = 300;
19     private static final int PREFERRED_HEIGHT = 200;
20
21     public void start(Stage stage)
22     {
23         Text message = new Text(MESSAGE_X, MESSAGE_Y,
24             "Not a Hello World program");
25
26         Pane root = new Pane(message);
27         root.setPrefSize(PREFERRED_WIDTH, PREFERRED_HEIGHT);
28
29         Scene scene = new Scene(root);
30         stage.setScene(scene);
31         stage.setTitle("NotHelloWorld");
32         stage.show();
33     }
34 }
```



NOTE: As you see from this example, no `main` method is required to launch a JavaFX application. The `java` program launcher knows about JavaFX and calls its `launch` method.

In previous versions of JavaFX, you were required to include a `main` method of the form

```
public class MyApp extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }
    . . .
}
```

You can still do this if your tool chain is flustered by an absence of public static `void main`.

javafx.stage.Stage

- `void setScene(Scene value)`
sets the scene to be shown on this stage.
- `void setTitle(String value)`
sets the title that is shown in the window's title bar.
- `void show()`
shows the window.

javafx.scene.layout.Pane

- `Pane(Node... children)`
constructs a pane holding the given child nodes.

javafx.scene.layout.Region

- `void setPrefSize(double prefWidth, double prefHeight)`
sets the preferred size of this region to the given width and height.

javafx.scene.text.Text

- `Text(double x, double y, String text)`
constructs a Text node with the given position and contents.

13.2.2 Drawing Shapes

In JavaFX, geometric shapes are subclasses of the `Shape` class, itself a subclass of `Node`. To draw an image made up of rectangles, lines, circles, and other shapes, you simply construct the shapes and then construct a root node containing the shapes:

```
Rectangle rect = new Rectangle(leftX, topY, width, height);  
Line line = new Line(centerX, centerY, centerX + radius, centerY);  
Pane root = new Pane(rect, line);
```

If you need to add a node afterwards, call the `getChildren` method of the root pane, which yields a mutable `List<Node>`. By adding or removing nodes, you can update the children of the pane.

```
Circle circle = new Circle(centerX, centerY, radius);
root.getChildren().add(circle);
```



NOTE: Object-oriented design purists complain that methods such as `getChildren` violate the “Law of Demeter” since they give out mutable innards of an object. But this is common practice in JavaFX.



NOTE: In JavaFX, you construct circles and ellipses from the center points and radii. This is different (and more convenient) than with AWT and Swing, where you need to specify the bounding rectangle.



NOTE: To draw shapes in Swing or Android, you need to place drawing operations into a `paintComponent` or `onDraw` callback. The JavaFX API is much simpler. You simply add the nodes that you want to be drawn to the scene. If you move the nodes, the scene gets automatically redrawn.

Listing 13.2 draw/DrawTest.java

```
1 package draw;
2
3 import javafx.application.*;
4 import javafx.scene.*;
5 import javafx.scene.layout.*;
6 import javafx.scene.paint.*;
7 import javafx.scene.shape.*;
8 import javafx.stage.*;
9
10 /**
11  * @version 1.4 2017-12-23
12  * @author Cay Horstmann
13  */
14 public class DrawTest extends Application
15 {
16     private static final int PREFERRED_WIDTH = 400;
17     private static final int PREFERRED_HEIGHT = 400;
18
19     public void start(Stage stage)
20     {
21         double leftX = 100;
22         double topY = 100;
```



```
23     double width = 200;
24     double height = 150;
25
26     Rectangle rect = new Rectangle(leftX, topY, width, height);
27     rect.setFill(Color.TRANSPARENT);
28     rect.setStroke(Color.BLACK);
29     // an ellipse touching the rectangle
30     double centerX = leftX + width / 2;
31     double centerY = topY + height / 2;
32     Ellipse ellipse = new Ellipse(centerX, centerY, width / 2, height / 2);
33     ellipse.setFill(Color.PEACHPUFF);
34     // a diagonal line
35     Line diagonal = new Line(leftX, topY, leftX + width, topY + height);
36     // a circle with the same center as the ellipse
37     double radius = 150;
38     Circle circle = new Circle(centerX, centerY, radius);
39     circle.setFill(Color.TRANSPARENT);
40     circle.setStroke(Color.RED);
41     Pane root = new Pane(rect, ellipse, diagonal, circle);
42     root.setPrefSize(PREFERRED_WIDTH, PREFERRED_HEIGHT);
43     stage.setScene(new Scene(root));
44     stage.setTitle("DrawTest");
45     stage.show();
46 }
47 }
```

Nodes of type `Line`, `Path`, and `Polygon` are by default drawn in black. For a different color, call the `setStroke` method:

```
radius.setStroke(Color.RED);
```

Shapes other than `Line`, `Path`, and `Polygon` are filled with a black color. You can change the fill color:

```
rect.setFill(Color.YELLOW);
```

Or, if you don't want the interior of the shape colored, choose a transparent fill. Then you need to set a stroke color for the shape's outline:

```
rect.setFill(Color.TRANSPARENT);
rect.setStroke(Color.BLACK);
```

The `setFill` and `setStroke` methods accept a `Paint` parameter type. The `Color` class is a subclass of `Paint`, as are classes for gradients and image patterns that we do not discuss here. There are predefined constants for all 147 CSS3 color names from `Color.ALICEBLUE` to `Color.YELLOWGREEN`.

Listing 13.2 contains a program that draws the shapes shown in Figure 13.4.

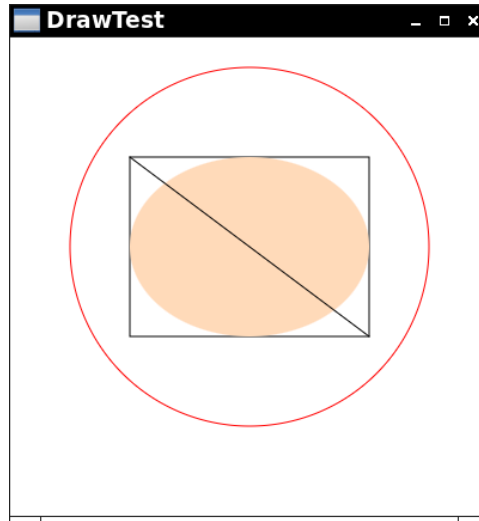


Figure 13.4 Drawing geometric shapes

`javafx.scene.shape.Rectangle`

- `Rectangle(double x, double y, double width, double height)`
constructs a rectangle with the given top left corner, width, and height.

`javafx.scene.shape.Circle`

- `Circle(double centerX, double centerY, double radius)`
constructs a circle with the given center and radius.

`javafx.scene.shape.Ellipse`

- `Ellipse(double centerX, double centerY, double radiusX, double radiusY)`
constructs an ellipse with the given center and radii.

`javafx.scene.shape.Line`

- `Line(double startX, double startY, double endX, double endY)`
constructs a line with the given start and end points.

class javafx.scene.layout.Pane

- `ObservableList<Node> getChildren()`
yields a mutable list of all children of this pane.

javafx.scene.shape.Shape

- `void setStroke(Paint value)`
sets the paint for drawing the boundary of this shape, or in the case of `Line`, `Polyline`, and `Path`, the shape itself.
- `void setFill(Paint value)`
sets the paint for drawing the interior of this shape.

13.2.3 Text and Images

The “Not a Hello World” program at the beginning of this chapter displayed a string in the “System” font at its default size. Often, you will want to show your text in a different font. Use one of the static `Font.font` methods to obtain the font, and then call the `setFont` method on the `Text` object to set the font.

```
message.setFont(Font.font("Times New Roman", 36));
```

This `Font` factory method makes a font object representing the font with the given family name and point size. You can specify a bold and italic version by calling

```
Font.font("Times New Roman", FontWeight.BOLD, FontPosture.ITALIC, 36);
```

Fonts for the family names

```
System  
Serif  
SansSerif  
Monospaced
```

are always available. The JDK ships with three font families:

```
Lucida Bright  
Lucida Sans  
Lucida Sans Typewriter
```

The static method `Font.getFamilies` yields a list of all available family names.



CAUTION: Any number of fonts may share a given family name. For example, the Lucida Bright family has members named Lucida Bright Regular, Lucida Bright Demibold, and Lucida Bright Demibold Italic. These names are of limited utility since the JavaFX API does not allow you to choose a font by its name.

To further confuse matters, the `FontWeight` enumeration has values `THIN`, `EXTRA_LIGHT`, `LIGHT`, `NORMAL`, `MEDIUM`, `SEMI_BOLD`, `BOLD`, `EXTRA_BOLD`, and `BLACK`, and you are on your own trying to map a string such as “Demibold” to a supported weight.

The *y*-position of a `Text` node indicates the *baseline* of the text (see Figure 13.5). To find the extent of the text, call

```
Bounds messageBounds = message.getBoundsInParent();
```

You can then compute the ascent (the distance from the baseline to the top of a letter such as ‘b’ or ‘k’) and the descent (the distance from the baseline to the bottom of a letter such as ‘p’ or ‘q’):

```
double ascent = message.getY() - messageBounds.getMinY();  
double descent = messageBounds.getMaxY() - message.getY();  
double width = messageBounds.getWidth();
```



CAUTION: The `Node` class has three methods to determine the bounds of a node: `getLayoutBounds`, `getBoundsInLocal`, and `getBoundsInParent`. Only the `getBoundsInParent` method takes stroke widths, effects, and transforms into account. Use that method whenever you want to know the extent of a node as it is actually drawn.

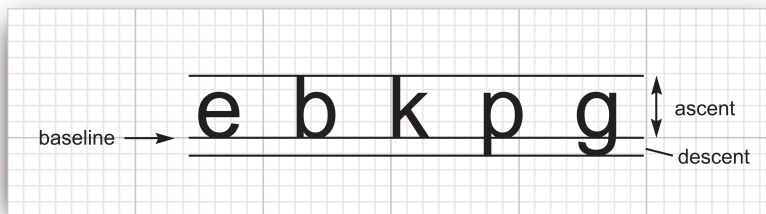


Figure 13.5 Typesetting terms illustrated

The program in Listing 13.3 shows how to accurately position a `Text` node. We construct the text at the origin, and measure its ascent, descent, and width. Then we center the text horizontally and place the baseline at the desired position, using the `relocate` method of the `Node` class. That method



Figure 13.6 Drawing the baseline and text bounds

relocates the top left corner, not the base point, and we need to adjust the y -position by the ascent.

To show that all the computations are accurate, we draw the bounding rectangle and the baseline (see Figure 13.6). We use the French version of “Hello, World!” so that the message contains a letter with a descender.

Next, we place an image directly below the text. To add an image, construct an `ImageView` from the image path or URL. You cannot specify the top left corner in the constructor. Therefore, we use the `relocate` method to move the image view.

Listing 13.3 font/FontTest.java

```
1 package font;
2
3 import javafx.application.*;
4 import javafx.geometry.*;
5 import javafx.scene.*;
6 import javafx.scene.image.*;
7 import javafx.scene.layout.*;
8 import javafx.scene.paint.*;
9 import javafx.scene.shape.*;
10 import javafx.scene.text.*;
11 import javafx.stage.*;
```

(Continues)

Listing 13.3 *(Continued)*

```
12
13 /**
14     @version 1.4 2017-12-23
15     @author Cay Horstmann
16 */
17 public class FontTest extends Application
18 {
19     private static final int PREFERRED_WIDTH = 400;
20     private static final int PREFERRED_HEIGHT = 400;
21
22     public void start(Stage stage)
23     {
24         // construct message at (0, 0)
25         Text message = new Text("Bonjour le monde!");
26         Font f = Font.font("Lucida Bright", FontWeight.BOLD, 36);
27         message.setFont(f);
28
29         // get message dimensions
30         Bounds messageBounds = message.getBoundsInParent();
31         double ascent = -messageBounds.getMinY();
32         double descent = messageBounds.getMaxY();
33         double width = messageBounds.getWidth();
34
35         // center message horizontally
36         double baseY = 100;
37         double topY = baseY - ascent;
38         double leftX = (PREFERRED_WIDTH - width) / 2;
39         message.relocate(leftX, topY);
40
41         // construct bounding rectangle and baseline
42         Rectangle rect = new Rectangle(leftX, topY, width, ascent + descent);
43         rect.setFill(Color.TRANSPARENT);
44         rect.setStroke(Color.GRAY);
45         Line baseline = new Line(leftX, baseY, leftX + width, baseY);
46         baseline.setStroke(Color.GRAY);
47
48         // center image directly below the message
49         ImageView image = new ImageView("font/world.png");
50         Bounds imageBounds = image.getBoundsInParent();
51         image.relocate((PREFERRED_WIDTH - imageBounds.getWidth()) / 2, baseY + descent);
52
53         Pane root = new Pane(message, rect, baseline, image);
54         root.setPrefSize(PREFERRED_WIDTH, PREFERRED_HEIGHT);
55         stage.setScene(new Scene(root));
56         stage.setTitle("FontTest");
57         stage.show();
58     }
59 }
```

javafx.scene.text.Font

- static Font.font(double size)
 - static Font.font(String family)
 - static Font.font(String family, double size)
 - static Font.font(String family, FontWeight weight, double size)
 - static Font.font(String family, FontPosture posture, double size)
 - static Font.font(String family, FontWeight weight, FontPosture posture, double size)
- obtains a font with the given family name (or “System”), weight and posture (or FontWeight.NORMAL and FontPosture.REGULAR), and point size.

javafx.scene.text.Text

- void setFont(Font value)
sets this text to the given font.
 - double getX()
 - double getY()
- gets the *x*- and *y*-position of the basepoint of this text node.

javafx.scene.Node

- Bounds getBoundsInParent()
gets the bounds of this node after applying any strokes, clips, effects, and transformations.
- void relocate(double x, double y)
relocates this node so that its top left corner falls on the given *x*- and *y*-values.

javafx.geometry.Bounds

- double getMinX()
 - double getMinY()
 - double getMaxX()
 - double getMaxY()
- gets the smallest or largest *x*- and *y*-value of these bounds.
- double getWidth()
 - double getHeight()
- gets the width and height of these bounds.

javafx.scene.image.ImageView

- `ImageView(String url)`
constructs an image from the given url string. The string should either be a valid construction parameter for the `java.net.URL` class, or a path to a resource. (See Chapter 5 about resources.)

13.3 Event Handling

A graphical user interface environment monitors input devices for events such as keystrokes or mouse clicks and directs them to the appropriate program. The program then figures out which user interface control should process the event, translating low-level events to semantic events as appropriate. For example, when a user clicks on a button, JavaFX processes the sequence of events that consists of depressing and releasing the mouse button over the surface of the button control. That event sequence is then interpreted as a “click.”

In order for the program to react to such an event, the programmer needs to register an *event handler* with the user interface control from which the event originates.

13.3.1 Implementing Event Handlers

In the case of a button click, the event handler must implement the `EventHandler<ActionEvent>` interface. `EventHandler<T>` is a functional interface with a single method

```
void handle(T event)
```

You can simply use a lambda expression to specify the button action:

```
Button button = new Button("Click me!");  
button.setOnAction(event -> System.out.println("I was clicked."));
```

In this case, the lambda expression did not make use of the event parameter. An `ActionEvent` doesn’t have many interesting properties. The most useful one is probably the event source—the control from which the action originated. In this case, we know which button is the source. However, if you share one handler among multiple controls, you can call `event.getSource()` to find out which one triggered it.

Here is another application of event handling where you are interested in the event object. When the user closes a window, the window gets a “close

request” event. You can install a handler in which you *consume* the event if you want to deny the request:

```
stage.setOnCloseRequest(event ->
{
    if (not OK to close) event.consume()
});
```



NOTE: If you need to add more than one event to a node, use the `addEventHandler` method:

```
button.addEventHandler(javafx.event.ActionEvent.ACTION,
    event -> System.out.println("I was clicked"));
```

13.3.2 Reacting to Property Changes

Many JavaFX controls provide a different mechanism for event handling. Consider a slider, as shown in Figure 13.7. When the slider is adjusted, its value changes. However, you shouldn’t listen to the low-level events that the slider emits to indicate those changes. Instead, the slider has a JavaFX *property* called `value`, and the property emits events when it changes. We will discuss properties in detail in Section 13.6, “Properties and Bindings,” on p. 82. Here is how you can listen to the property’s events and adjust the font size of a message:

```
slider.valueProperty().addListener(property ->
    message.setFont(Font.font(family, slider.getValue())));
```

Here, the property parameter is not very useful—it is easier to get the updated value from the slider.

Listening to properties is very common in JavaFX. For example, if you want to change a part of the user interface as a user enters text into a text field, simply add a listener to the text property.

The program in Listing 13.4 shows action and property change events at work. When you click on the “Random font” button, a `Text` node is set to the name of a random font and displayed in the same font—see Figure 13.7.

When the slider moves, the font size is adjusted.

When the user closes the window, the event listener checks whether the slider is at 100%. If so, the program refuses to close, assuming that the user hasn’t yet tried out the slider.

The button, slider, and text node are stacked vertically inside a `VBox`. We discuss this class in Section 13.4, “Layout,” on p. 28.



Figure 13.7 Processing action and property change events

Listing 13.4 event/EventTest.java

```

1 package event;
2
3 import java.util.*;
4
5 import javafx.application.*;
6 import javafx.scene.*;
7 import javafx.scene.control.*;
8 import javafx.scene.control.Alert.*;
9 import javafx.scene.layout.*;
10 import javafx.scene.text.*;
11 import javafx.stage.*;
12
13 /**
14  * @version 1.0 2017-12-23
15  * @author Cay Horstmann
16  */
17 public class EventTest extends Application
18 {
19     public void start(Stage stage)
20     {
21         Button button = new Button("Random font");
22         Text message = new Text("Times New Roman");
23         message.setFont(Font.font("Gloucester MT Extra Condensed", 100));
24         List<String> families = Font.getFamilies();
25         Random generator = new Random();
26         button.setOnAction(event ->
27             {
28                 String newFamily = families.get(
29                     generator.nextInt(families.size()));
30                 message.setText(newFamily);
31                 message.setFont(Font.font(
32                     newFamily, message.getFont().getSize()));
33             });
34
35         Slider slider = new Slider();
36         slider.setValue(100);

```

```

37     slider.valueProperty().addListener(property ->
38     {
39         double newSize = slider.getValue();
40         message.setFont(Font.font(
41             message.getFont().getFamily(), newSize));
42     });
43
44     VBox root = new VBox(button, slider, message);
45     Scene scene = new Scene(root);
46
47     stage.setTitle("EventTest");
48     stage.setScene(scene);
49     stage.setOnCloseRequest(event ->
50     {
51         if (slider.getValue() == 100)
52         {
53             event.consume(); // Stops window from closing
54             Alert alert = new Alert(AlertType.INFORMATION,
55                 "Move the slider before quitting.");
56             alert.showAndWait();
57         }
58     });
59     stage.show();
60 }
61 }

```

EventHandler<T extends Event>

- void handle(T event)
override this method to handle the given event.

javafx.scene.control.ButtonBase

- void setOnAction(EventHandler<ActionEvent> value)
sets the action event listener for this control.

javafx.event.Event

- void consume()
marks this event as consumed.
- boolean isConsumed()
returns true if this event has been marked as consumed.

java.util.EventObject 1.1

- Object getSource()
gets the object responsible for emitting this event.

javafx.stage.Window

- public final void setOnCloseRequest(EventHandler<WindowEvent> value)
sets the close request handler for this window. The handler should consume the event to reject the close request.

13.3.3 Mouse and Keyboard Events

You do not need to handle mouse events explicitly if you just want the user to be able to click on a button or drag a slider. These mouse operations are handled internally by the various controls in the user interface. However, if you want to enable the user to draw with the mouse, you will need to trap the events that happen as the user moves the mouse and clicks the mouse buttons.

In this section, we will show you a simple graphics editor application that allows the user to place, move, and erase dots on a canvas (see Figure 13.8).

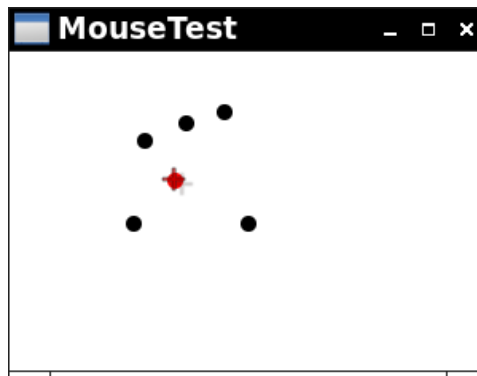


Figure 13.8 Arranging dots with the mouse and keyboard

When the user clicks a mouse button, three events are generated: “mouse pressed,” when the mouse button is first pressed; “mouse released,” when the mouse button is released; and, finally, “mouse clicked,” after pressing and releasing the button. In our example, we capture mouse presses because we don’t want to delay the visual feedback until the mouse button is released.

By using the `getX` and `getY` methods on the `MouseEvent` argument, you can obtain the *x*- and *y*-coordinates of the mouse pointer.

```
root.setOnMousePressed(event ->
{
    double x = event.getX();
    double y = event.getY();
    Circle dot = new Circle(x, y, RADIUS);
    root.getChildren().add(dot);
});
```

To distinguish between single, double, and triple (!) clicks, use the `getClickCount` method. To find out which button was pressed, call the `getButton` method:

```
if (event.getButton() == MouseButton.SECONDARY) . . . // right click
```

Some user interface designers inflict mouse click and keyboard modifier combinations, such as `Control+Shift+click`, on their users. We find this practice reprehensible, but if you disagree, call one of the following methods:

```
isShiftDown
isControlDown
isAltDown
isMetaDown
isShortcutDown
```




















The `Alt` key is labeled `Option` (⌘) on the Mac. The `Meta` key is the `Windows` or `Command` (⌘) key. The `isShortcutDown` method tests for the preferred modifier of the platform—`Control` on `Linux` and `Windows`, `Meta` on the `Mac`.

As the mouse moves, a steady stream of mouse movement events are generated. Any node can ask to be notified when the mouse passes over it. Our test application traps mouse motion events to change the cursor to a different shape (a cross hair) when it is over a dot:

```
dot.setOnMouseEntered(event -> scene.setCursor(Cursor.CROSSHAIR));
dot.setOnMouseExited(event -> scene.setCursor(Cursor.DEFAULT));
```

Table 10.3 lists the available cursors.

Table 13.1 Cursor Shapes

Icon	Constant	Icon	Constant
	DEFAULT		N_RESIZE
	CROSSHAIR		NE_RESIZE
	HAND		E_RESIZE
	OPEN_HAND		SE_RESIZE
	CLOSED_HAND		S_RESIZE
	MOVE		SW_RESIZE
	TEXT		W_RESIZE
	WAIT		NW_RESIZE
	DISAPPEAR		H_RESIZE
	NONE		V_RESIZE



NOTE: You can also define your own cursor types through the use of the `ImageCursor` class:

```
Image img = new Image("dynamite.gif");
Cursor dynamiteCursor = new ImageCursor(img, 10, 10);
```

The second and third parameters of the constructor give the offset of the “hot spot” of the cursor that represents the mouse position.

If the user presses a mouse button while the mouse is in motion, “mouse dragged” events are generated instead of “mouse moved” events. Our test application lets a user drag the dot under the cursor. We simply update the dot to be centered under the mouse position.

```
dot.setOnMouseDragged(event ->
{
    dot.setCenterX(event.getX());
    dot.setCenterY(event.getY());
});
```

A mouse event is sent to the node under the mouse cursor. But if a key event occurs, which node should be notified? The recipient is a specific node that has the *keyboard focus*. A node can request keyboard focus with the `requestFocus` method.

Similar to clicking a mouse button, key events occur as a key is pressed or released. When pressing and releasing one or more keys yields text input, a “key typed” event is generated. For example, if the user presses the Shift key, then presses and releases the A key, an uppercase ‘A’ is typed.

```
source.setOnKeyTyped(event ->
{
    String input = event.getCharacter();
    . . .
});
```

The `getCharacter` method gets the input as a string (in case some keyboard produces emoji that require two UTF-16 code units, or decomposed accents, or some other input that doesn’t fit in a single `char` value).

However, if you process cursor or function keys, you want to listen to “key pressed” events and call the `getCode` method. You get a value of the `KeyCode` enumeration, which has over 200 values with every imaginable key that can appear on someone’s keyboard, such as `KeyCode.A`, `KeyCode.DELETE`, and `KeyCode.EURO_SIGN`.

```
source.setOnKeyPressed(event ->
{
    KeyCode code = event.getCode();
    if (code == KeyCode.DELETE) . . . ;
});
```

The program in Listing 13.5 demonstrates mouse and key events. Use the mouse to click on an empty space to add a new dot, or drag an existing dot to a new location. If you double-click on a dot, it is erased. The dot with keyboard focus is colored red. You can delete it with the Delete key and move it with the cursor keys. If you hold down the Shift key, it moves by a greater distance.

Note that each dot listens to “mouse entered/exited/pressed/dragged” and “key pressed” events. We simply rely on the JavaFX framework to deliver the events to the affected nodes. The “mouse pressed” event is also handled by the root pane so that we can add new dots.

A mouse event is first presented to child nodes and then to parent nodes. To avoid creating another dot, the “mouse pressed” handler of the dot consumes the event.

Alternatively, we could have implemented a single “mouse pressed” handler for the root pane. But then we would have had to check if a node is under the mouse pointer. The JavaFX way is to work as much as possible with the “scene graph” (the nodes and their parent/child relationships).

Listing 13.5 mouse/MouseTest.java

```
1 package mouse;
2
3 import javafx.application.*;
4 import javafx.scene.*;
5 import javafx.scene.input.*;
6 import javafx.scene.layout.*;
7 import javafx.scene.paint.*;
8 import javafx.scene.shape.*;
9 import javafx.stage.*;
10
11 /**
12  * @version 1.40 2017-12-27
13  * @author Cay Horstmann
14  */
15 public class MouseTest extends Application
16 {
17     private static final int PREFERRED_WIDTH = 300;
18     private static final int PREFERRED_HEIGHT = 200;
19     private static final int RADIUS = 5;
20     private Scene scene;
21     private Pane root;
22     private Circle selected;
23
24     private Circle makeDot(double x, double y)
25     {
26         Circle dot = new Circle(x, y, RADIUS);
27         dot.setOnMouseEntered(event ->
28             scene.setCursor(Cursor.CROSSHAIR));
29         dot.setOnMouseExited(event ->
30             scene.setCursor(Cursor.DEFAULT));
31         dot.setOnMouseDragged(event ->
32             {
33                 dot.setCenterX(event.getX());
34                 dot.setCenterY(event.getY());
35             });
36         dot.setOnMousePressed(event ->
37             {
38                 if (event.getClickCount() > 1)
39                 {
40                     root.getChildren().remove(selected);
41                     select(null);
42                 }
43                 else
44                 {
45                     select(dot);
46                 }
47                 event.consume();
48             });
49     }
```



```
49
50     dot.setOnKeyPressed(event ->
51     {
52         KeyCode code = event.getCode();
53         int distance = event.isShiftDown() ? 10 : 1;
54         if (code == KeyCode.DELETE)
55             root.getChildren().remove(dot);
56         else if (code == KeyCode.UP)
57             dot.setCenterY(dot.getCenterY() - distance);
58         else if (code == KeyCode.DOWN)
59             dot.setCenterY(dot.getCenterY() + distance);
60         else if (code == KeyCode.LEFT)
61             dot.setCenterX(dot.getCenterX() - distance);
62         else if (code == KeyCode.RIGHT)
63             dot.setCenterX(dot.getCenterX() + distance);
64     });
65
66     return dot;
67 }
68
69 private void select(Circle dot)
70 {
71     if (selected == dot) return;
72     if (selected != null) selected.setFill(Color.BLACK);
73     selected = dot;
74     if (selected != null)
75     {
76         selected.requestFocus();
77         selected.setFill(Color.RED);
78     }
79 }
80
81 public void start(Stage stage)
82 {
83     root = new Pane();
84     root.setOnMousePressed(event ->
85     {
86         double x = event.getX();
87         double y = event.getY();
88         Circle dot = makeDot(x, y);
89         root.getChildren().add(dot);
90         select(dot);
91     });
92     scene = new Scene(root);
93     root.setPrefSize(PREFERRED_WIDTH, PREFERRED_HEIGHT);
94     stage.setScene(scene);
95     stage.setTitle("MouseTest");
96     stage.show();
97 }
98 }
```

javafx.scene.Node

- void setOnMousePressed(EventHandler<? super MouseEvent> value)
- void setOnMouseReleased(EventHandler<? super MouseEvent> value)
- void setOnMouseClicked(EventHandler<? super MouseEvent> value)
- void setOnMouseEntered(EventHandler<? super MouseEvent> value)
- void setOnMouseExited(EventHandler<? super MouseEvent> value)
- void setOnMouseMoved(EventHandler<? super MouseEvent> value)
- void setOnMouseDragged(EventHandler<? super MouseEvent> value)

sets the handler for the given mouse event type.

- void setOnKeyPressed(EventHandler<? super KeyEvent> value)
- void setOnKeyReleased(EventHandler<? super KeyEvent> value)
- void setOnKeyTyped(EventHandler<? super KeyEvent> value)

sets the handler for the given key event type.

- void requestFocus()

requests that the keyboard focus be set to this node.

javafx.scene.input.MouseEvent

- double getX()
- double getY()

yields the x - and y -coordinate of the mouse pointer in the coordinate system of the event source.

- double getScreenX()
- double getScreenY()

yields the x - and y -coordinate of the mouse pointer in the screen coordinate system.

- int getClickCount()

gets the number of times a mouse button has been clicked (within a small region and a short amount of time).

- MouseButton getButton()

returns one of the values PRIMARY, SECONDARY, MIDDLE, or NONE of the MouseButton enumeration.

(Continues)

javafx.scene.input.MouseEvent *(Continued)*

- `boolean isShiftDown()`
- `boolean isControlDown()`
- `boolean isAltDown()`
- `boolean isMetaDown()`
- `boolean isShortcutDown()`

returns true if the Shift, Control, Alt/Option, Windows/Command, or Control/Command modifier key is pressed during this event.

javafx.scene.Scene

- `void setCursor(Cursor value)`

sets the mouse cursor for this scene. See Table 10.3 for predefined cursor shapes.

javafx.scene.ImageCursor

- `ImageCursor(Image image, double hotspotX, double hotspotY)`

constructs an image cursor from the given image. The cursor will be placed so that the given hotspot offset is at the cursor position.

javafx.scene.input.KeyEvent

- `String getCharacter()`
if this is a “key typed” event, gets the input that was typed.
- `KeyCode getCode()`
if this is a “key pressed” or “key released” event, gets the key that was pressed or released.
- `boolean isShiftDown()`
- `boolean isControlDown()`
- `boolean isAltDown()`
- `boolean isMetaDown()`
- `boolean isShortcutDown()`

returns true if the Shift, Control, Alt/Option, Windows/Command, or Control/Command modifier key is pressed during this event.

13.4 Layout

When a graphical user interface contains multiple controls, they need to be arranged on the screen in a functional and attractive way. One way to obtain a layout is with a design tool. The tool's user, often a graphics designer, drags images of the controls onto a design view and arranges, resizes, and configures them. However, this approach can be problematic when the sizes of the elements change—for example, because labels have different lengths in international versions of a program.

Alternatively, the layout can be achieved programmatically, by writing code in a setup method that adds the user interface controls to specific positions. That is what was done in Swing, using layout manager objects.

Another approach is to specify the layout in a declarative language. For example, web pages are laid out with HTML and CSS. Similarly, Android has an XML language for specifying layouts.

JavaFX supports all three approaches. The JavaFX Scene Builder is a visual GUI builder. You can download it from <http://gluonhq.com/products/scene-builder>. Figure 13.9 shows a screenshot.

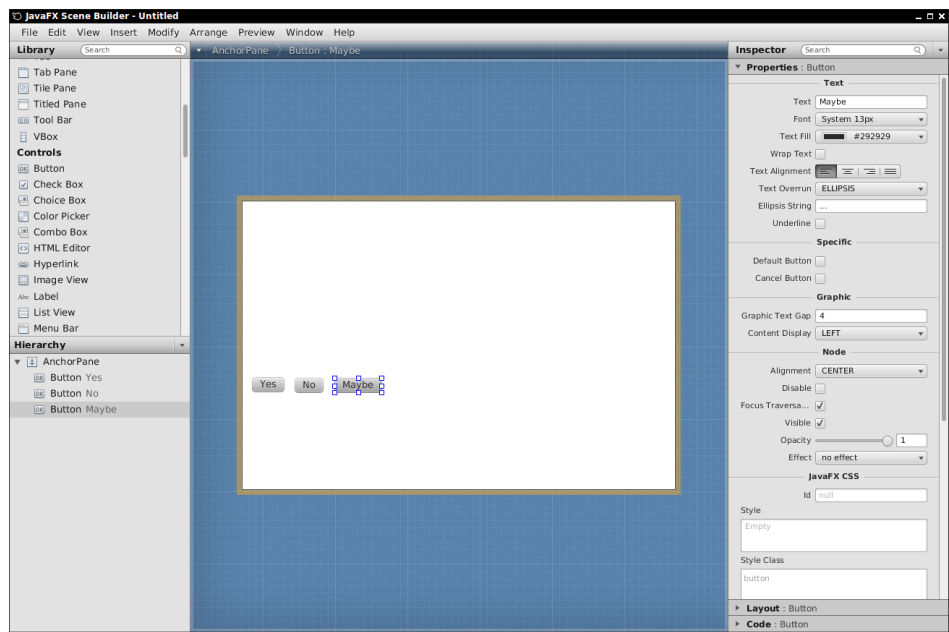


Figure 13.9 The JavaFX Scene Builder

We won't discuss the Scene Builder program further. When you understand the concepts of the following sections, you will find it straightforward to use. Read on for laying out elements with layout panes and with the FXML markup language.

13.4.1 Layout Panes

In JavaFX, you can arrange controls with *panes*—parent nodes with a layout policy. For example, a `BorderPane` has five areas: Top, Bottom, Left, Right, and Center. Here we place a button into each:

```
BorderPane pane = new BorderPane();
pane.setTop(new Button("Top"));
pane.setRight(new Button("Right"));
pane.setBottom(new Button("Bottom"));
pane.setLeft(new Button("Left"));
pane.setCenter(new Button("Center"));
stage.setScene(new Scene(pane));
```

Figure 13.10 shows the result.

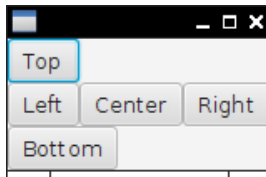


Figure 13.10 The `BorderPane` layout



NOTE: With the Swing `BorderLayout`, buttons were expanded to fill each region of the layout. In JavaFX, a button does not expand past its natural size.

Now suppose you want more than one button in the bottom area. Use an `HBox` (see Figure 13.11):

```
HBox buttons = new HBox(10, yesButton, noButton, maybeButton);
// ten pixels between controls
pane.setBottom(buttons);
```

Of course, there is a `VBox` for laying out controls vertically. The layout in Figure 13.11 was achieved like this:

```
VBox pane = new VBox(10, question, buttons);
pane.setPadding(new Insets(10));
```

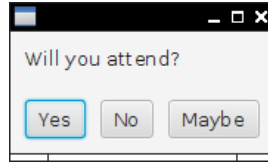


Figure 13.11 Laying out buttons with an HBox

Note the padding property. Without it, the label and the buttons would touch the window border.



CAUTION: In JavaFX, dimensions are specified in pixels. In our example, we use ten pixels for the box spacing and padding. This is not really appropriate nowadays, when pixel densities can vary widely. One way to overcome this is to compute dimensions in rem, as you would do in CSS3. (A rem or “root em” is the height of the default font of the document root.)

```
final double rem = new Text("").getBoundsInParent().getHeight();  
pane.setPadding(new Insets(0.8 * rem));
```

There is only so much you can achieve with horizontal and vertical boxes. Just as Swing had the `GridBagLayout` as “the mother of all layout managers,” JavaFX has the `GridPane`. Think of a `GridPane` as an equivalent of an HTML table. You can set the horizontal and vertical alignment of all cells. If desired, cells can span multiple rows and columns. Consider the login dialog in Figure 13.12.

Note the following:

- The labels “User name:” and “Password:” are right-aligned.
- The buttons are in an `HBox` that spans two columns.

When you add a child to a `GridPane`, specify its column and row index (in that order; think *x*- and *y*-coordinates).

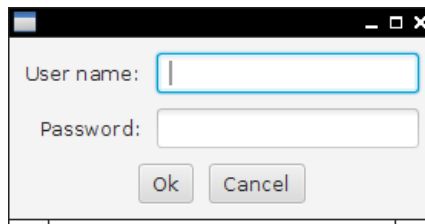


Figure 13.12 A `GridPane` can arrange the controls for this login dialog.

```
pane.add(usernameLabel, 0, 0);
pane.add(username, 1, 0);
pane.add(passwordLabel, 0, 1);
pane.add(password, 1, 1);
```

If a child spans multiple columns or rows, specify the spans after the positions. For example, the button panel spans two columns and one row:

```
pane.add(buttons, 0, 2, 2, 1);
```

If you want a child to span all remaining rows or columns, use `GridPane.REMAINING`.

To set the horizontal alignment of a child, use the static `setHalignment` method, and pass the child reference and a constant `LEFT`, `CENTER`, or `RIGHT` from the `HPos` enumeration.

```
GridPane.setHalignment(usernameLabel, HPos.RIGHT);
```

Similarly, for vertical alignment, call `setValignment` and use `TOP`, `CENTER`, `BASELINE`, or `BOTTOM` from the `VPos` enumeration.



NOTE: These static calls look rather inelegant in Java code, but they make sense in the FXML markup language—see the next section.



TIP: For debugging, it can be useful to see the cell boundaries (see Figure 13.13). Call

```
pane.setGridLinesVisible(true);
```

If you want to see the borders of an individual child (for example, to see whether it has grown to fill the entire cell), set its border. This is most easily done with CSS:

```
buttons.setStyle("-fx-border-color: red;");
```

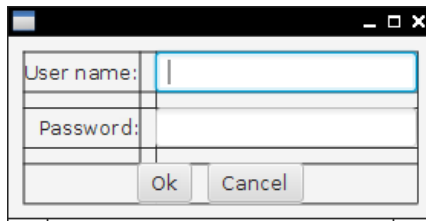


Figure 13.13 Use visible grid lines when debugging a `GridPane`.



CAUTION: Do *not* center the HBox with the buttons inside the grid. That box has expanded to the full horizontal size, and centering will not change its position. Instead, tell the HBox to center its contents:

```
buttons.setAlignment(Pos.CENTER);
```

You will also want to provide some spacing around the rows and columns and some padding around the table:

```
pane.setHgap(0.8 * rem);
pane.setVgap(0.8 * rem);
pane.setPadding(new Insets(0.8 * rem));
```

The layout panes that you have seen in this section should suffice for the majority of applications. Table 13.2 shows all layouts that come with JavaFX.

Listing 13.6 shows the complete program for this layout. For more information about the user interface controls that the program uses, turn to Section 13.5.1, “Text Input,” on p. 47.

Table 13.2 JavaFX Layouts

Pane Class	Description
HBox, VBox	Lines up children horizontally or vertically.
GridPane	Lays out children in a tabular grid, similar to the Swing GridBagLayout.
TilePane	Lays out children in a grid, giving them all the same size, similar to the Swing GridLayout.
BorderPane	Provides the areas North, East, South, West, and Center, similar to the Swing BorderLayout.
FlowPane	Flows children in rows, making new rows when there isn’t sufficient space, similar to the Swing FlowLayout.
AnchorPane	Places children in absolute positions or relative to the pane’s boundaries. This is the default in the Scene Builder layout tool.
StackPane	Stacks children above each other. Can be useful for decorating controls, such as stacking a button over a colored rectangle.



TIP: Scala, Kotlin, and Groovy provide JavaFX bindings (<http://www.scalafx.org>, <http://tornadofx.io>, <http://groovyfx.org>) with convenient “domain-specific languages” for building user interfaces that are reminiscent of the old FX Script builder syntax.

Listing 13.6 gridPane/GridPaneDemo.java

```
1 package gridPane;
2
3 import javafx.application.*;
4 import javafx.geometry.*;
5 import javafx.scene.*;
6 import javafx.scene.control.*;
7 import javafx.scene.layout.*;
8 import javafx.scene.text.*;
9 import javafx.stage.*;
10
11 public class GridPaneDemo extends Application
12 {
13     public void start(Stage stage)
14     {
15         final double rem = new Text("").getLayoutBounds().getHeight();
16
17         GridPane pane = new GridPane();
18         // uncomment for debugging
19         // pane.setGridLinesVisible(true);
20
21         pane.setHgap(0.8 * rem);
22         pane.setVgap(0.8 * rem);
23         pane.setPadding(new Insets(0.8 * rem));
24         Label usernameLabel = new Label("User name:");
25         Label passwordLabel = new Label("Password:");
26         TextField username = new TextField();
27         PasswordField password = new PasswordField();
28
29         Button okButton = new Button("Ok");
30         Button cancelButton = new Button("Cancel");
31
32         HBox buttons = new HBox(0.8 * rem);
33         buttons.getChildren().addAll(okButton, cancelButton);
34         buttons.setAlignment(Pos.CENTER);
35         // uncomment for debugging
36         // buttons.setStyle("-fx-border-color: red;");
37
38         pane.add(usernameLabel, 0, 0);
39         pane.add(username, 1, 0);
40         pane.add(passwordLabel, 0, 1);
41         pane.add(password, 1, 1);
42         pane.add(buttons, 0, 2, 2, 1);
43
44         GridPane.setHalignment(usernameLabel, HPos.RIGHT);
45         GridPane.setHalignment(passwordLabel, HPos.RIGHT);
```

(Continues)

Listing 13.6 *(Continued)*

```
46     stage.setScene(new Scene(pane));
47     stage.show();
48 }
49 }
```

javafx.scene.layout.BorderPane

- `BorderPane()`
constructs an empty border pane.
- `void setTop(Node value)`
- `void setRight(Node value)`
- `void setBottom(Node value)`
- `void setLeft(Node value)`
- `void setCenter(Node value)`
places a node into an area of this border pane.

javafx.scene.layout.HBox

- `HBox(double spacing, Node... children)`
constructs a horizontal box with the given children, separated by the number of pixels given in spacing.
- `void setAlignment(Pos pos)`
sets the alignment for the children. The `Pos` enumeration has values `TOP_LEFT`, `TOP_CENTER`, `TOP_RIGHT`, `CENTER_LEFT`, `CENTER`, `CENTER_RIGHT`, `BASELINE_LEFT`, `BASELINE_CENTER`, `BASELINE_RIGHT`, `BOTTOM_LEFT`, `BOTTOM_CENTER`, `BOTTOM_RIGHT`.

javafx.scene.layout.VBox

- `VBox(double spacing, Node... children)`
constructs a vertical box with the given children, separated by the number of pixels given in spacing.

class javafx.scene.layout.Region

- `void setPadding(Insets value)`
sets the padding around the contents of this region.

javafx.geometry.Insets

- `public Insets(double topRightBottomLeft)`
 - `public Insets(double top, double right, double bottom, double left)`
- constructs insets with the given number of pixels to the top, right, bottom, and left.

javafx.scene.layout.GridPane

- `GridPane()`
constructs an empty grid pane.
- `void add(Node child, int columnIndex, int rowIndex)`
- `void add(Node child, int columnIndex, int rowIndex, int colspan, int rowspan)`
adds a node at the given position. The second method has the control occupy multiple grid cells. To occupy all remaining rows or columns, pass `GridPane.REMAINING`.
- `static void setHalignment(Node child, HPos value)`
sets the horizontal alignment of the given node within its grid cell. The `HPos` enumeration has values `LEFT`, `RIGHT`, and `CENTER`.
- `static void setValignment(Node child, VPos value)`
sets the vertical alignment of the given node within its grid cell. The `VPos` enumeration has values `TOP`, `CENTER`, `BASELINE`, and `BOTTOM`.
- `void setHgap(double value)`
- `void setVgap(double value)`
sets the horizontal or vertical gap between rows or columns to the given number of pixels.

13.4.2 FXML

The markup language that JavaFX uses to describe layouts is called FXML. We discuss it in some detail because the concepts are interesting beyond the needs of JavaFX, and the implementation is fairly general.

Here is the FXML markup for the login dialog of the preceding section:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
```

```

<?import javafx.scene.text.*?>
<?import javafx.scene.layout.*?>

<GridPane hgap="10" vgap="10">
  <padding>
    <Insets top="10" right="10" bottom="10" left="10"/>
  </padding>
  <children>
    <Label text="User name:" GridPane.columnIndex="0" GridPane.rowIndex="0"
      GridPane.halignment="RIGHT"/>
    <Label text="Password:" GridPane.columnIndex="0" GridPane.rowIndex="1"
      GridPane.halignment="RIGHT"/>
    <TextField GridPane.columnIndex="1" GridPane.rowIndex="0"/>
    <PasswordField GridPane.columnIndex="1" GridPane.rowIndex="1"/>
    <HBox GridPane.columnIndex="0" GridPane.rowIndex="2"
      GridPane.columnSpan="2" alignment="CENTER" spacing="10">
      <children>
        <Button text="Ok"/>
        <Button text="Cancel"/>
      </children>
    </HBox>
  </children>
</GridPane>

```

Have a closer look at the FXML file. Note the “processing instructions” `<?import . . . ?>` for importing Java packages. (In general, XML processing instructions are an “escape hatch” for application-specific processing of XML documents.)

Now look at the structure of the document. First off, the nesting of the `GridPane`, the labels and text fields, the `HBox` and its button children reflects the nesting that we built up with Java code in the preceding section.

Most of the attributes correspond to property setters. For example,

```
<GridPane hgap="10" vgap="10">
```

means “construct a `GridPane` and then set the `hgap` and `vgap` properties.”

When an attribute starts with a class name and a static method, that method is invoked. For example,

```
<TextField GridPane.columnIndex="1" GridPane.rowIndex="0"/>
```

means that the static methods `GridPane.setColumnIndex(thisTextField, 1)` and `GridPane.setRowIndex(thisTextField, 0)` will be called.

When a property value is too complex to express as a string, one uses nested elements instead of attributes. Consider, for example,

```
<GridPane hgap="10" vgap="10">
  <padding>
    <Insets top="10" right="10" bottom="10" left="10"/>
  </padding>
  . . .
```

The padding property has type Insets, and the Insets object is constructed with an `<Insets . . .>` child element that specifies how to set its properties.

Finally, there is a special rule for list properties. For example, `children` is a list property. Calling

```
<HBox . . .>
  <children>
    <Button text="Ok" />
    <Button text="Cancel" />
  </children>
</HBox>
```

adds the buttons to the list returned by `getChildren`.



NOTE: You can localize text strings by using resource keys starting with %, such as `<Button text="%ok">`. Then you must provide a resource bundle that maps resource keys to localized values. We discuss resource bundles in Chapter 7 of Volume II.

You can write FXML files by hand, or you can use a GUI builder such as Scene Builder. Once you have such a file, load it like this:

```
public void start(Stage stage)
{
    try
    {
        Parent root = FXMLLoader.load(getClass().getResource("dialog.fxml"));
        stage.setScene(new Scene(root));
        stage.show();
    }
    catch (IOException e)
    {
        e.printStackTrace();
        System.exit(0);
    }
}
```

Of course, this is not yet useful by itself. The user interface is displayed, but the program cannot access the values that the user provides. One way of establishing a connection between the controls and the program is to use `id` attributes, as you would in JavaScript. Provide the `id` attributes in the FXML file:

```
<TextField id="username" GridPane.columnIndex="1" GridPane.rowIndex="0"/>
```

In the program, look up the control:

```
TextField username = (TextField) root.lookup("#username");
```

But there is a better way. You can use the `@FXML` annotation to “inject” the control objects into a *controller* class. The controller class must implement the `Initializable` interface. In the controller’s `initialize` method, you wire up the event handlers. Any class can be the controller, even the FX application itself.

For example, here is a controller for our login dialog:

```
public class LoginDialogController implements Initializable
{
    @FXML private TextField username;
    @FXML private PasswordField password;
    @FXML private Button okButton;
    @FXML private Button cancelButton;

    public void initialize(URL url, ResourceBundle rb)
    {
        okButton.setOnAction(event -> . . .);
        cancelButton.setOnAction(event ->
        {
            username.setText("");
            password.setText("");
        });
    }
}
```

Provide the names of the controller’s instance variables to the corresponding control elements in the FXML file, using the `fx:id` (not `id`) attribute:

```
<TextField fx:id="username" GridPane.columnIndex="1" GridPane.rowIndex="0"/>
<PasswordField fx:id="password" GridPane.columnIndex="1" GridPane.rowIndex="1" />
<Button fx:id="okButton" text="Ok" />
```

In the root element, you also need to declare the controller class, using the `fx:controller` attribute:

```
<GridPane xmlns:fx="http://javafx.com/fxml" hgap="10" vgap="10"
    fx:controller="LoginDialogController">
```

Note the namespace attribute to introduce the FXML namespace.

When the FXML file is loaded, the scene graph is constructed, and references to the named control objects are injected into the annotated fields of the controller object. Then its `initialize` method is called.



NOTE: If your controller doesn't have a default constructor (perhaps because it is being initialized with a reference to a business service), you can set it programmatically:

```
FXMLLoader loader = new FXMLLoader(getClass().getResource(. . .));  
loader.setController(new Controller(service));  
Parent root = loader.load();
```



CAUTION: If you set the controller programmatically, really use the code from the preceding note. The following code will compile, but it will invoke the static `FXMLLoader.load` method, ignoring the constructed loader:

```
FXMLLoader loader = new FXMLLoader();  
loader.setController(new Controller(service));  
Parent root = loader.load(getClass().getResource(. . .));  
// ERROR--calls static method
```

It is even possible to do much of the initialization in the FXML file. You can define simple bindings, and you can set annotated controller methods as event listeners. It is also possible to add scripts in JavaScript or another scripting language to an FXML file. The syntax is documented at https://docs.oracle.com/javase/9/docs/api/javafx/fxml/doc-files/introduction_to_fxml.html. However, let's not dwell on these features. It seems better to separate the visual design from the program behavior, so that a user interface designer can produce the design and a programmer can implement the behavior.

Listing 13.7 fxml/FXMLDemo.java

```
1 package fxml;  
2  
3 import java.io.*;  
4 import java.net.*;  
5 import java.util.*;  
6  
7 import javafx.application.*;  
8 import javafx.fxml.*;  
9 import javafx.scene.*;  
10 import javafx.scene.control.*;
```

(Continues)

Listing 13.7 *(Continued)*

```
11 import javafx.scene.control.Alert.*;
12 import javafx.stage.*;
13
14 /**
15     @version 1.0 2017-12-29
16     @author Cay Horstmann
17 */
18 public class FXMLDemo extends Application implements Initializable
19 {
20     @FXML private TextField username;
21     @FXML private PasswordField password;
22     @FXML private Button okButton;
23     @FXML private Button cancelButton;
24
25     public void initialize(URL url, ResourceBundle rb)
26     {
27         okButton.setOnAction(event ->
28         {
29             Alert alert = new Alert(AlertType.INFORMATION,
30                 "Verifying " + username.getText() + ":" + password.getText());
31             alert.showAndWait();
32         });
33         cancelButton.setOnAction(event ->
34         {
35             username.setText("");
36             password.setText("");
37         });
38     }
39
40     public void start(Stage stage)
41     {
42         try
43         {
44             Parent root = FXMLLoader.load(
45                 getClass().getResource("dialog.fxml"));
46             stage.setScene(new Scene(root));
47             stage.setTitle("FXMLDemo");
48             stage.show();
49         }
50         catch (IOException e)
51         {
52             e.printStackTrace();
53         }
54     }
55 }
```

Listing 13.8 fxml/dialog.fxml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <?import java.lang.*?>
4 <?import java.util.*?>
5 <?import javafx.geometry.*?>
6 <?import javafx.scene.control.*?>
7 <?import javafx.scene.text.*?>
8 <?import javafx.scene.layout.*?>
9
10 <GridPane xmlns:fx="http://javafx.com/fxml" hgap="10" vgap="10"
11     fx:controller="FXMLDemo">
12     <padding>
13         <Insets top="10" right="10" bottom="10" left="10"/>
14     </padding>
15     <children>
16         <Label text="User name:" GridPane.columnIndex="0"
17             GridPane.rowIndex="0" GridPane.halignment="RIGHT"/>
18         <Label text="Password:" GridPane.columnIndex="0"
19             GridPane.rowIndex="1" GridPane.halignment="RIGHT"/>
20         <TextField fx:id="username" GridPane.columnIndex="1"
21             GridPane.rowIndex="0"/>
22         <PasswordField fx:id="password"
23             GridPane.columnIndex="1" GridPane.rowIndex="1"/>
24         <HBox GridPane.columnIndex="0" GridPane.rowIndex="2"
25             GridPane.columnSpan="2" alignment="CENTER" spacing="10">
26             <children>
27                 <Button fx:id="okButton" text="Ok"/>
28                 <Button fx:id="cancelButton" text="Cancel"/>
29             </children>
30         </HBox>
31     </children>
32 </GridPane>

```

javafx.fxml.FXMLLoader

- static <T> T load(URL location)
- static <T> T load(URL location, ResourceBundle resources)

returns the object described by the FXML document at the given location. The second method uses the given resource bundle for resolving resource keys prefixed by %.

(Continues)

javafx.fxml.FXMLLoader *(Continued)*

- `FXMLLoader(URL location)`
constructs a loader that will load an object from the FXML document at the given location, using the given resource bundle for resolving resource keys.
- `void setController(Object controller)`
sets the controller for the root element. This method must be called before loading the FXML document.
- `<T> T load()`
returns the object described by the FXML document of this loader.

javafx.fxml.Initializable

- `void initialize(URL location, ResourceBundle resources)`
This method is called when the FXMLLoader has constructed the root element and its controller. The location and resources are those of the loader.

13.4.3 CSS

JavaFX lets you change the visual appearance of the user interface with CSS, which is usually more convenient than supplying FXML attributes or calling Java methods.

You can load a CSS stylesheet programmatically and have it applied to a scene graph:

```
Scene scene = new Scene(pane);
scene.getStylesheets().add("scene.css");
```

In the stylesheet, you can reference any controls that have an ID. For example, here is how you can control the appearance of a `GridPane`. In the code, set the ID:

```
GridPane pane = new GridPane();
pane.setId("pane");
```

Don't set any padding or spacing in the code. Instead, use CSS.

```
#pane {
    -fx-padding: 0.5em;
    -fx-hgap: 0.5em;
    -fx-vgap: 0.5em;
```

```
-fx-background-image: url("metal.jpg")
}
```

Unfortunately, you can't use the familiar CSS attributes but need to know FX-specific attributes that start with `-fx-`. The attribute names are formed by changing the property names to lowercase and using hyphens instead of camel case. For example, the `textAlignment` property turns into `-fx-text-alignment`. You can find all supported attributes in the JavaFX CSS reference at <https://docs.oracle.com/javase/9/docs/api/javafx/scene/doc-files/cssref.html>.

Using CSS is nicer than cluttering up the code with layout minutiae. Moreover, you can easily use resolution-independent `em` units. Of course, CSS can be used both for good and for evil (see Figure 13.14), and hopefully you will resist the temptation to apply gratuitous background textures to your login dialogs.

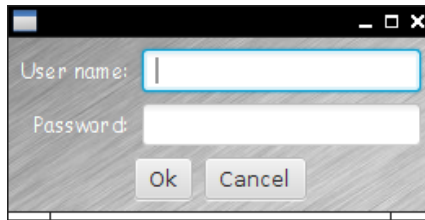


Figure 13.14 Using CSS to style a user interface

Instead of styling by individual IDs, you can use style classes. Add the class to the node object:

```
HBox buttons = new HBox();
buttons.getStyleClass().add("buttonrow");
```

Then style it, using the CSS class notation:

```
.buttonrow {
    -fx-spacing: 0.5em;
}
```

Every JavaFX control and shape class belongs to a CSS class whose name is the decapitalized Java class name. For example, all `Label` nodes have class `label`. Here is how you can change the font for all labels to Comic Sans:

```
.label {
    -fx-font-family: "Comic Sans MS";
}
```

But please don't.

You can also use CSS with FXML layouts. Attach the stylesheet to the root pane:

```
<GridPane id="pane" stylesheets="scene.css">
```

Supply `id` or `styleClass` attributes in the FXML code. For example,

```
<HBox styleClass="buttonrow">
```

Then you can specify most styling in CSS, and use FXML only for layout. Unfortunately, you can't completely remove all styling from the FXML. For example, there is currently no way to specify grid cell alignment in CSS.



NOTE: You can also apply a CSS style programmatically, such as

```
buttons.setStyle("-fx-border-color: red;");
```

That can be handy for debugging, but in general, it's better to use an external stylesheet.

The program in Listing 13.9 demonstrates the use of stylesheets. The stylesheet in Listing 13.10 is loaded directly by the program. The second stylesheet (Listing 13.11) is loaded by the FXML file in Listing 13.12.

Listing 13.9 `css/CSSDemo.java`

```
1 package css;
2
3 import java.io.*;
4 import javafx.application.*;
5 import javafx.fxml.*;
6 import javafx.scene.*;
7 import javafx.scene.control.*;
8 import javafx.stage.*;
9
10 public class CSSDemo extends Application
11 {
12     public void start(Stage stage)
13     {
14         try
15         {
16             Parent root = FXMLLoader.load(getClass().getResource("dialog.fxml"));
17             root.lookup("#username").getStyleClass().add("highlight");
18             Scene scene = new Scene(root);
19             scene.getStylesheets().add("css/scene1.css");
```

```
20         stage.setScene(scene);
21         stage.setTitle("CSSDemo");
22         stage.show();
23     }
24     catch (IOException ex)
25     {
26         ex.printStackTrace();
27         Platform.exit();
28     }
29 }
30 }
```

Listing 13.10 css/scene1.css

```
1 .label {
2     -fx-text-fill: white;
3     -fx-font-family: "Comic Sans MS";
4 }
5
6 #pane {
7     -fx-padding: 0.5em;
8     -fx-hgap: 0.5em;
9     -fx-vgap: 0.5em;
10    -fx-background-image: url("metal.jpg");
11 }
12
13 .highlight:focused {
14     -fx-border-color: yellow;
15 }
```

Listing 13.11 css/scene2.css

```
1 .buttonrow {
2     -fx-spacing: 0.5em;
3     -fx-alignment: center;
4 }
5
```

Listing 13.12 css/dialog.fxml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <?import java.lang.*?>
4 <?import java.util.*?>
5 <?import javafx.geometry.*?>
```

(Continues)

Listing 13.12 *(Continued)*

```

6 <?import javafx.scene.control.*?>
7 <?import javafx.scene.text.*?>
8 <?import javafx.scene.layout.*?>
9
10 <GridPane id="pane" xmlns:fx="http://javafx.com/fxml" stylesheets="css/scene2.css">
11     <children>
12         <Label text="User name:" GridPane.columnIndex="0" GridPane.rowIndex="0"
13             GridPane.halignment="RIGHT"/>
14         <Label text="Password: " GridPane.columnIndex="0" GridPane.rowIndex="1"
15             GridPane.halignment="RIGHT"/>
16         <TextField id="username" GridPane.columnIndex="1" GridPane.rowIndex="0"/>
17         <PasswordField GridPane.columnIndex="1" GridPane.rowIndex="1"/>
18         <HBox styleClass="buttonrow" GridPane.columnIndex="0" GridPane.rowIndex="2"
19             GridPane.columnSpan="2">
20             <children>
21                 <Button text="Ok"/>
22                 <Button text="Cancel"/>
23             </children>
24         </HBox>
25     </children>
26 </GridPane>

```

javafx.scene.Scene

- `ObservableList<String> getStylesheets()`
returns the list of URL strings for the CSS stylesheets of this scene.

javafx.scene.Node

- `void setId(String value)`
sets an ID for this node. The ID can be used in CSS stylesheets.
- `void setStyle(String value)`
sets the CSS style of this node.

javafx.css.Styleable

- `ObservableList<String> getStyleClass()`
returns the list of CSS style class names for this node.

13.5 User Interface Controls

In the following sections, we go over common user interface controls: text controls, checkboxes, radio buttons, combo boxes, menus, and simple dialogs.

13.5.1 Text Input

We begin with the controls that let a user input and edit text. A `TextField` can accept only one line of text; a `TextArea` accepts multiple lines of text. As you can see in Figure 13.15, these are both subclasses of `TextInputControl`, which is itself a subclass of `Control`, the ancestor of all user interface controls in JavaFX. The `JPasswordField` class is a subclass of `TextField` and accepts one line of text without showing the contents.

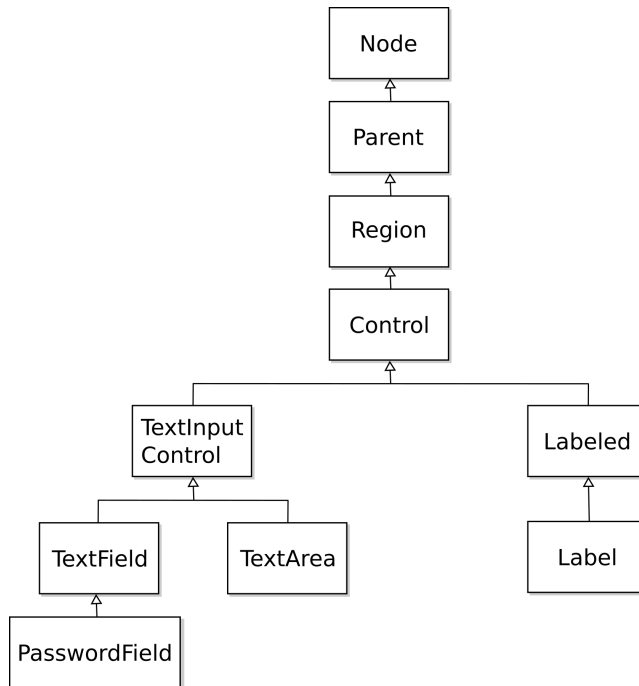


Figure 13.15 The inheritance hierarchy of text input controls

The `TextInputControl` class has methods `getText`, `setText`, and `appendText` for getting and setting the text, or appending to it.

By calling the `setPrefColumnCount` method of a `TextField`, you give a hint for the preferred size. One column is the expected width of one character in the font you are using for the text. The user can still type in longer strings, but the input scrolls when the text exceeds the width of the field. For text areas, there is also a `setPrefRowCount` method.



NOTE: Scrollbars automatically appear if there is more text than a text area can display, and they vanish again if text is deleted and the remaining text fits inside the area.

You can enable wrapping of long lines in a text area by calling

```
textArea.setWrapText(true);
```

Then the text area will not use horizontal scroll bars.

All text controls can be made read-only:

```
textArea.setEditable(false);
```

A read-only text area can be used to display a large amount of information. This does not change the appearance of the text area. In contrast, a disabled control has a distinct visual appearance that indicates its disabled status to the user.

If you set the *prompt text*, it is shown in gray if the control is empty (see Figure 13.16).

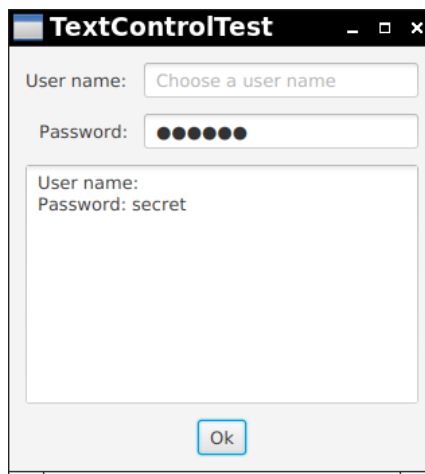


Figure 13.16 Text controls


```
username.setPromptText("Choose a user name");
```

Unlike buttons, text controls need to be labeled with an external `Label` instance that you place nearby:

```
Label usernameLabel = new Label("User name:");
```

A `Label` is similar to a `Text` node, but a `Text` is a `Shape`, meant to be used for drawing. In contrast, a `Label` is a `Control`. It inherits methods from the `Labeled` superclass for tasks such as setting a graphic decoration and controlling how the text should be truncated if there is insufficient space to display it.

Listing 13.13 demonstrates the various text controls. This program shows a text field, a password field, and a text area with scrollbars. The text field and password field are labeled. Click on “Ok” to insert the field contents into the text area.

Listing 13.13 `text/TextControlTest.java`

```
1 package text;
2
3 import javafx.application.*;
4 import javafx.geometry.*;
5 import javafx.scene.*;
6 import javafx.scene.control.*;
7 import javafx.scene.layout.*;
8 import javafx.scene.text.*;
9 import javafx.stage.*;
10
11 /**
12  * @version 1.5 2017-12-29
13  * @author Cay Horstmann
14  */
15 public class TextControlTest extends Application
16 {
17     public void start(Stage stage)
18     {
19         final double rem = new Text("").getLayoutBounds().getHeight();
20
21         GridPane pane = new GridPane();
22         pane.setHgap(0.8 * rem);
23         pane.setVgap(0.8 * rem);
24         pane.setPadding(new Insets(0.8 * rem));
25
26         Label usernameLabel = new Label("User name:");
27         Label passwordLabel = new Label("Password:");
28
```

(Continues)

Listing 13.13 *(Continued)*

```

29     TextField username = new TextField();
30     username.setPromptText("Choose a user name");
31     PasswordField password = new PasswordField();
32     password.setPromptText("Choose a password");
33     TextArea textArea = new TextArea();
34     textArea.setPrefRowCount(10);
35     textArea.setPrefColumnCount(20);
36     textArea.setWrapText(true);
37     textArea.setEditable(false);
38
39     Button okButton = new Button("Ok");
40     okButton.setOnAction(event ->
41         textArea.appendText("User name: " + username.getText()
42             + "\nPassword: " + password.getText() + "\n"));
43
44     pane.add(usernameLabel, 0, 0);
45     pane.add(username, 1, 0);
46     pane.add(passwordLabel, 0, 1);
47     pane.add(password, 1, 1);
48     pane.add(textArea, 0, 2, 2, 1);
49     pane.add(okButton, 0, 3, 2, 1);
50
51     GridPane.setHalignment(usernameLabel, HPos.RIGHT);
52     GridPane.setHalignment(passwordLabel, HPos.RIGHT);
53     GridPane.setHalignment(okButton, HPos.CENTER);
54
55     stage.setScene(new Scene(pane));
56     stage.setTitle("TextControlTest");
57     stage.show();
58 }
59 }
60 }

```

javafx.scene.Node

- `void setDisable(boolean value)`
disables or enables this node and its children. Subclasses of `Control` render themselves differently when they are disabled.

javafx.scene.control.TextInputControl

- `String getText()`
- `void setText(String value)`
gets or sets the text in this control.
- `void appendText(String text)`
appends the given text to the text in this control.
- `void setEditable(boolean value)`
makes this control editable or read-only.
- `void setPromptText(String value)`
sets the prompt text that is displayed in the control.

javafx.scene.control.TextField

- `void setPrefColumnCount(int value)`
sets the preferred column count of this text field.

javafx.scene.control.TextArea

- `void setPrefRowCount(int value)`
- `void setPrefColumnCount(int value)`
sets the preferred row and column count of this text area.
- `void setWrapText(boolean value)`
enables or disables text wrapping for lines whose length exceeds the width of this text area.

javafx.scene.control.Label

- `Label(String text)`
constructs a label with the given text.

javafx.scene.control.Labeled

- `void setGraphic(Node value)`
adds a decorative node to this labeled control.
- `void setTextOverrun(OverrunStyle value)`
sets the text overrun policy to indicate where the ellipses (...) should appear when the label text needs to be truncated: not at all; at the beginning, middle or end; at character or word boundaries. Use one of the following values of the `OverrunStyle` enumeration: `CLIP`, `ELLIPSIS`, `WORD_ELLIPSIS`, `LEADING_ELLIPSIS`, `LEADING_WORD_ELLIPSIS`, `CENTER_ELLIPSIS`, `CENTER_WORD_ELLIPSIS`.

13.5.2 Choices

You now know how to collect text input from users, but there are many occasions where you would rather give users a finite set of choices than have them enter the data in a text control. Using radio buttons or a list of items tells your users what choices they have. It also saves you the trouble of error checking. In this section, you will learn how to program checkboxes, radio buttons, and combo boxes.

In our example program, you will be able to choose the font weight, posture, size, and family, as shown in Figure 13.17.

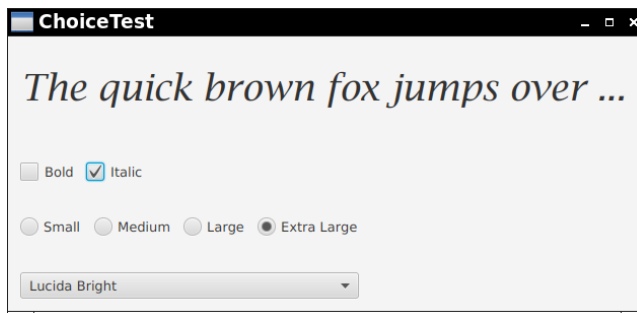


Figure 13.17 Choice controls for selecting a font

If you want to collect just a “yes” or “no” input, use a checkbox control. For example, a font is italic, or it isn’t. Checkboxes come with labels that identify them. Give the label text in the constructor:

```
CheckBox italic = new CheckBox("Italic");
```

The user can check the box by clicking inside it and turn off the checkmark by clicking inside the box again. Pressing the space bar when the focus is in the checkbox also toggles the checkmark.

Use the `setSelected` method to turn a checkbox on or off. For example:

```
italic.setSelected(true);
```

The `isSelected` method retrieves the current state of a checkbox. It is `false` if unchecked, `true` if checked.

When the user clicks on a checkbox, this triggers an action event. As always, you call the `setOnAction` method to attach an event handler to the checkbox. The `CheckBox` class inherits this method from the `ButtonBase` superclass.

In our example program, we offer checkboxes for bold and italic fonts. The user can check either, both, or neither of the two checkboxes.

In many cases, we want the user to check only one of several boxes. When another box is checked, the previous box is automatically unchecked. Such a group of boxes is called a radio button group because the buttons work like the station selector buttons on an old-fashioned radio. When you push in one button, the previously depressed button pops out. (Admittedly, that analogy made much more sense thirty years ago than it does now. But these controls are still called radio buttons.)

A typical example is the font size selection in our sample program. We allow the user to select a font size from among four choices—Small, Medium, Large, or Extra Large—but, of course, we will allow selecting only one size at a time.

Constructing a radio button is straightforward:

```
RadioButton small = new RadioButton("Small");
```

To get the desired behavior of having the previously selected button turn off when a new one is selected, all buttons need to be in the same *toggle group*:

```
ToggleGroup group = new ToggleGroup();
small.setToggleGroup(group);
medium.setToggleGroup(group);
. . .
```

You also want to make sure that one of the buttons is selected at the outset:

```
medium.setSelected(true);
```

As with checkboxes, radio buttons send action events when they are clicked. You could attach a separate event handler to each radio button, but that gets repetitive. We will share a single handler, but we want to avoid code that

checks whether the event source was the Small, Medium, Large, or Extra Large button. All we care about is the desired size of the font.

User data provide a convenient mechanism for transferring arbitrary data from a control to a shared handler. You can set arbitrary objects as user data of a control. We will set the desired font sizes:

```
small.setUserData(8);
medium.setUserData(14);
. . .
```

In the shared event handler, we retrieve the user data from the currently selected button in the toggle group:

```
int size = (int) group.getSelectedToggle().getUserData();
```

If you look again at Figure 13.17, you will note that the appearance of the radio buttons is different from that of checkboxes. Checkboxes are square and contain a checkmark when selected. Radio buttons are round and contain a dot when selected.

If you have more than a handful of alternatives, radio buttons are not a good choice because they take up too much screen space. Instead, use a combo box. When the user clicks on this control, a list of choices drops down, and the user can then select one of them (see Figure 13.18).

If the drop-down list box is set to be editable, you can edit the current selection as if it were a text field. For that reason, this control is called a combo box—it combines the flexibility of a text field with a set of predefined choices. The `ComboBox` class provides a combo box control.

Call the `setEditable` method to make the combo box editable. Note that editing affects only the selected item. It does not change the list of choices in any way.

In the example program, the user can choose a font family from a list of all available families. We add all choices to the list returned by `getItems`:

```
ComboBox<String> families = new ComboBox<>();
families.getItems().addAll(Font.getFamilies());
```



NOTE: `ComboBox<T>` is a generic class. The type parameter denotes the type of the items. If the item type is not `String`, then a converter needs to convert between the item type and the strings that the user views and potentially edits. We take up that issue in Chapter 11 of Volume II.

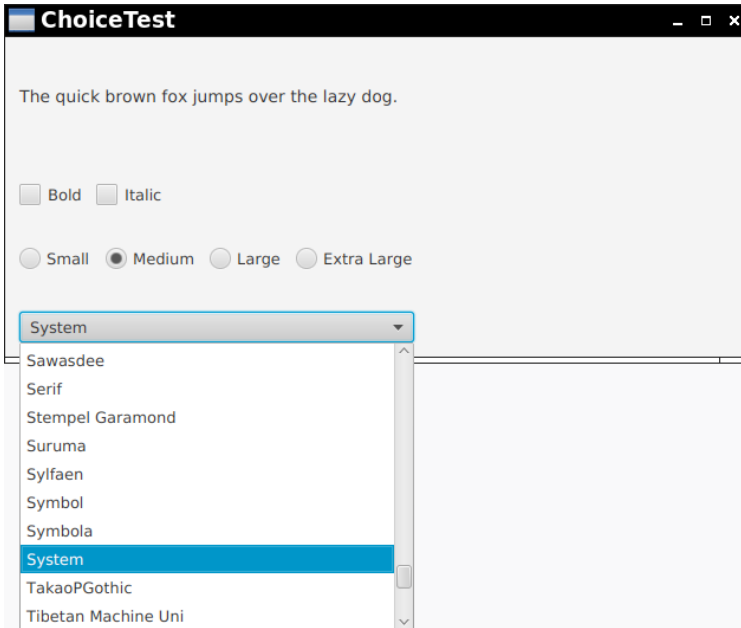


Figure 13.18 A combo box

You can obtain the current selection, which may have been edited if the combo box is editable, by calling the `getValue` method. Call `setValue` to specify an initial value.

As with the other choice controls, a `ComboBox` fires an action event when the user makes a selection. In our sample program, shown in Listing 13.14, it is not worth having separate event handlers for each control. Instead, all controls share a single handler that determines a font from the selected values in the checkboxes, radio buttons, and combo box.

Listing 13.14 choices/ChoiceTest.java

```
1 package choices;
2
3 import javafx.application.*;
4 import javafx.event.*;
5 import javafx.geometry.*;
6 import javafx.scene.*;
7 import javafx.scene.control.*;
8 import javafx.scene.layout.*;
```

(Continues)

Listing 13.14 *(Continued)*

```
9 import javafx.scene.text.*;
10 import javafx.stage.*;
11
12 /**
13     @version 1.4 2017-12-29
14     @author Cay Horstmann
15 */
16 public class ChoiceTest extends Application
17 {
18     private static final double rem = new Text("").getLayoutBounds().getHeight();
19
20     private static HBox hbox(Node... children)
21     {
22         HBox box = new HBox(0.8 * rem, children);
23         box.setPadding(new Insets(0.8 * rem));
24         return box;
25     }
26
27     public void start(Stage stage)
28     {
29         Label sampleText = new Label("The quick brown fox jumps over the lazy dog.");
30         sampleText.setPrefWidth(40 * rem);
31         sampleText.setPrefHeight(5 * rem);
32         sampleText.setFont(Font.font(14));
33
34         CheckBox bold = new CheckBox("Bold");
35         CheckBox italic = new CheckBox("Italic");
36
37         RadioButton small = new RadioButton("Small");
38         RadioButton medium = new RadioButton("Medium");
39         RadioButton large = new RadioButton("Large");
40         RadioButton extraLarge = new RadioButton("Extra Large");
41
42         small.setUserData(8);
43         medium.setUserData(14);
44         large.setUserData(18);
45         extraLarge.setUserData(36);
46
47         ToggleGroup group = new ToggleGroup();
48         small.setToggleGroup(group);
49         medium.setToggleGroup(group);
50         large.setToggleGroup(group);
51         extraLarge.setToggleGroup(group);
52         medium.setSelected(true);
53
54         ComboBox<String> families = new ComboBox<>();
55         families.getItems().addAll(Font.getFamilies());
```



```

56     families.setValue("System");
57
58     EventHandler<ActionEvent> listener = event ->
59     {
60         int size = (int) group.getSelectedToggle().getUserData();
61         Font font = Font.font(
62             families.getValue(),
63             bold.isSelected() ? FontWeight.BOLD : FontWeight.NORMAL,
64             italic.isSelected() ? FontPosture.ITALIC : FontPosture.REGULAR,
65             size);
66         sampleText.setFont(font);
67     };
68     small.setOnAction(listener);
69     medium.setOnAction(listener);
70     large.setOnAction(listener);
71     extraLarge.setOnAction(listener);
72     bold.setOnAction(listener);
73     italic.setOnAction(listener);
74     families.setOnAction(listener);
75
76     VBox root = new VBox(0.8 * rem,
77         hbox(sampleText),
78         hbox(bold, italic),
79         hbox(small, medium, large, extraLarge),
80         hbox(families));
81
82     stage.setScene(new Scene(root));
83     stage.setTitle("ChoiceTest");
84     stage.show();
85 }
86 }

```

javafx.scene.control.CheckBox

- `CheckBox(String text)`
constructs a checkbox labeled with the given text.
- `boolean isSelected()`
- `void setSelected(boolean value)`
gets or sets the selected state of this checkbox.

javafx.scene.control.RadioButton

- `RadioButton(String text)`
constructs a radio button labeled with the given text.

javafx.scene.control.ToggleButton

- void setToggleGroup(ToggleGroup value)
sets the toggle group for this button.
- boolean isSelected()
- void setSelected(boolean value)
gets or sets the selected state of this button.

javafx.scene.control.ToggleGroup

- ToggleGroup()
constructs an empty toggle group.
- Toggle getSelectedToggle()
gets the selected item from this toggle group.

javafx.scene.control.Toggle

- void setUserData(Object value)
- Object getUserData()
sets and gets a data item that is associated with this toggle control.

javafx.scene.control.ComboBox<T>

- ComboBox()
constructs a combo box with no items.
- ObservableList<T> getItems()
yields a mutable list of items in this combo box.

javafx.scene.control.ComboBoxBase<T>

- void setValue(T)
- T getValue()
sets or gets the currently selected value.
- void setOnAction(EventHandler<ActionEvent>)
sets the action event listener for this control.

13.5.3 Menus

We started by introducing the most common controls that you might want to place into a window, such as various kinds of buttons, text fields, and combo boxes. JavaFX also supports another type of user interface element—pull-down menus that are familiar from GUI applications.

A *menu bar* at the top of a window contains the names of the pull-down menus. Clicking on a name opens the menu containing menu items and submenus. When the user clicks on a menu item, an action event is triggered. Figure 13.19 shows a typical menu with a submenu.



Figure 13.19 A menu with a submenu

You construct a menu item by providing the menu text and, optionally, a graphical decoration. The decoration can be any *Node*, but it is most common to provide an image.

```
MenuItem newItem = new MenuItem("New");
MenuItem cutItem = new MenuItem("Cut", new ImageView("menu/cut.gif"));
```

You construct a *Menu* object in the same way.

```
Menu editMenu = new Menu("Edit");
```

Add menu items, separators, and submenus to a menu object by adding them to the list of items:

```
editMenu.getItems().addAll(cutItem, copyItem, pasteItem);
```

Alternatively, you can provide the children when you construct the menu:

```
Menu optionsMenu = new Menu("Options", new ImageView("menu/options.gif"),
    readOnlyItem, insertItem, overtypeItem);
```

If you don't want a decoration, supply *null* as the second argument.

Add the top-level menus to a *MenuBar*, and add the menu bar to the root pane:

```
MenuBar bar = new MenuBar(fileMenu, editMenu, helpMenu);
VBox root = new VBox(bar, . . .);
```

When the user selects a menu, an action event is triggered. You need to install an event handler for each menu item:

```
exitItem.setOnAction(event -> Platform.exit());
```

`CheckMenuItem` and `RadioMenuItem` controls display a check mark next to the item. They work exactly like checkboxes and radio buttons. You must add radio buttons to a toggle group. When one of the items in a group is selected, all others are automatically deselected.

```
CheckMenuItem readOnlyItem = new CheckMenuItem("Read-only");

ToggleGroup group = new ToggleGroup();
RadioMenuItem insertItem = new RadioMenuItem("Insert");
insertItem.setToggleGroup(group);
insertItem.setSelected(true);
RadioMenuItem overtypeItem = new RadioMenuItem("Overtyping");
overtimeItem.setToggleGroup(group);
```

In our sample application, checking the “Read-only” item deactivates the “Save” and “Save as” menu items. You can achieve this by installing an event listener:

```
readOnlyItem.setOnAction(event ->
{
    saveItem.setDisable(readOnlyItem.isSelected());
    saveAsItem.setDisable(readOnlyItem.isSelected());
});
```

Alternatively, as you will see in Section 13.6.2, “Bindings,” on p. 84, you can use a more elegant approach and bind properties:

```
saveItem.disableProperty().bind(readOnlyItem.selectedProperty());
saveAsItem.disableProperty().bind(readOnlyItem.selectedProperty());
```

A disabled menu item is grayed out—see Figure 13.20.

Accelerators are keyboard shortcuts that let you select menu items without ever opening a menu. For example, many programs attach the accelerators `Control+O` and `Control+S` (or `Command+O` and `Command+S` on the Mac) to the Open and Save items in the File menu. Use the `setAccelerator` method to attach an accelerator key to a menu item.

```
openItem.setAccelerator(KeyCombination.keyCombination("Shortcut+O"));
```



Figure 13.20 Disabled menu items

The “Shortcut” modifier denotes the Control key on Windows/Linux and the Command key on the Mac. The accelerator is displayed next to the menu item—see Figure 13.21.

Typing the accelerator key combination automatically selects the menu option and fires an action event, as if the user had selected the menu option manually. You can attach accelerators only to menu items, not to menus.



Figure 13.21 Accelerators

There is a limited number of accelerators available, even if you add modifier keys. Besides, few users have the mental power to remember shortcuts like Control+Shift+F11. On Linux and Windows, *keyboard mnemonics* allow users to navigate menus with the keyboard by pressing the Alt key and typing underlined letters. For example, in the menu shown in Figure 13.22, hitting Alt+H C selects the Help menu and then the “About Core Java” item. Unlike keyboard shortcuts, mnemonics let you reach all menu items with key combinations that are fairly easy to remember.

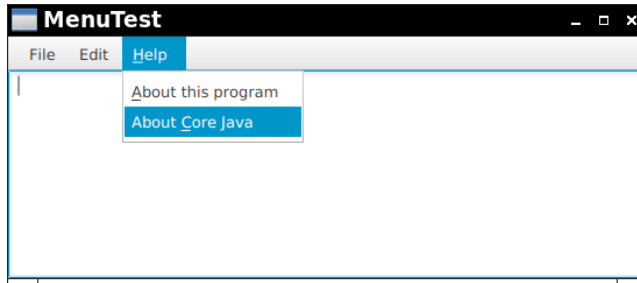


Figure 13.22 Keyboard mnemonics

You create a keyboard mnemonic for a menu or menu item by adding an underscore before the mnemonic letter in the menu item constructor:

```
MenuItem aboutProgramItem = new MenuItem("_About this program");
MenuItem aboutCoreJavaItem = new MenuItem("About _Core Java");
Menu helpMenu = new Menu("_Help", null, aboutProgramItem, aboutCoreJavaItem);
```



NOTE: In a misguided effort to reduce “visual noise,” most user interface environments (including JavaFX) hide mnemonics until you press the Alt key. MacOS doesn’t support mnemonics at all.

A *context menu* is a menu that is not attached to a menu bar but floats somewhere (see Figure 13.23). Create a context menu just as you create a regular menu, except that a context menu has no title. Then attach the context menu to a control. The context menu pops up when you right-click on the control. (On the Mac, hold down the Control key and click.)

```
ContextMenu contextMenu = new ContextMenu(cutItem, copyItem, pasteItem);
textArea.setContextMenu(contextMenu);
```

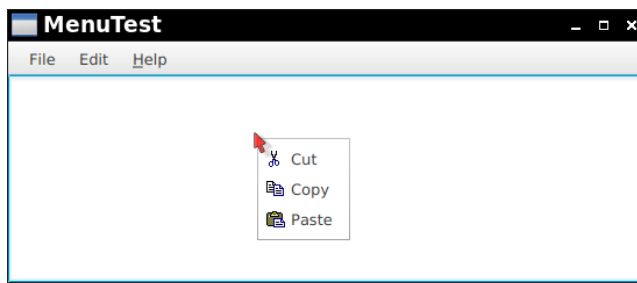


Figure 13.23 A context menu



NOTE: If you want a context menu to pop up over an arbitrary node, you have to work a bit harder and show the menu when the user right-clicks on the node.

```
node.setOnContextMenuRequested(e ->
    contextMenu.show(node, e.getScreenX(), e.getScreenY()));
```

The program in Listing 13.15 demonstrates all the features that you saw in this section: nested menus, disabled menu items, checkbox and radio menu items, a context menu, and keyboard mnemonics and accelerators.

Listing 13.15 menu/MenuTest.java

```
1 package menu;
2
3 import javafx.application.*;
4 import javafx.event.*;
5 import javafx.scene.*;
6 import javafx.scene.control.*;
7 import javafx.scene.image.*;
8 import javafx.scene.input.*;
9 import javafx.scene.layout.*;
10 import javafx.stage.*;
11
12 /**
13  * @version 1.3 2017-12-29
14  * @author Cay Horstmann
15  */
16 public class MenuTest extends Application
17 {
18     private TextArea textArea = new TextArea();
19
20     /**
21      * Makes this item or, if a menu, its descendant items, carry out the
22      * given action if there isn't already an action defined.
23      * @param item the menu item (which may be a menu)
24      * @param action the default action
25      */
26     private void defaultAction(MenuItem item, EventHandler<ActionEvent> action)
27     {
28         if (item instanceof Menu)
29             for (MenuItem child : ((Menu) item).getItems())
30                 defaultAction(child, action);
31         else if (item.getOnAction() == null)
32             item.setOnAction(action);
33     }
34 }
```

(Continues)

Listing 13.15 *(Continued)*

```
35 public void start(Stage stage)
36 {
37     Menu fileMenu = new Menu("File");
38     MenuItem exitItem = new MenuItem("Exit");
39     exitItem.setOnAction(event -> Platform.exit());
40
41     // demonstrate accelerators
42
43     MenuItem newItem = new MenuItem("New");
44     MenuItem openItem = new MenuItem("Open ...");
45     openItem.setAccelerator(KeyCombination.keyCombination("Shortcut+O"));
46     MenuItem saveItem = new MenuItem("Save");
47     saveItem.setAccelerator(KeyCombination.keyCombination("Shortcut+S"));
48     MenuItem saveAsItem = new MenuItem("Save as ...");
49
50     fileMenu.getItems().addAll(newItem,
51         openItem,
52         saveItem,
53         saveAsItem,
54         new SeparatorMenuItem(),
55         exitItem);
56
57     // demonstrate checkbox and radio button menus
58
59     CheckMenuItem readOnlyItem = new CheckMenuItem("Read-only");
60     readOnlyItem.setOnAction(event ->
61     {
62         saveItem.setDisable(readOnlyItem.isSelected());
63         saveAsItem.setDisable(readOnlyItem.isSelected());
64     });
65     /*
66     Or use binding:
67     saveItem.disableProperty().bind(readOnlyItem.selectedProperty());
68     saveAsItem.disableProperty().bind(readOnlyItem.selectedProperty());
69     */
70
71     ToggleGroup group = new ToggleGroup();
72     RadioMenuItem insertItem = new RadioMenuItem("Insert");
73     insertItem.setToggleGroup(group);
74     insertItem.setSelected(true);
75     RadioMenuItem overtypeItem = new RadioMenuItem("Overtypen");
76     overtypeItem.setToggleGroup(group);
77
78     Menu editMenu = new Menu("Edit");
79
```



```
80     // demonstrate icons
81
82     MenuItem cutItem = new MenuItem("Cut",
83         new ImageView("menu/cut.gif"));
84     MenuItem copyItem = new MenuItem("Copy",
85         new ImageView("menu/copy.gif"));
86     MenuItem pasteItem = new MenuItem("Paste",
87         new ImageView("menu/paste.gif"));
88
89     // demonstrate context menu
90
91     ContextMenu contextMenu = new ContextMenu(cutItem, copyItem, pasteItem);
92     textArea.setContextMenu(contextMenu);
93
94     editMenu.getItems().addAll(cutItem, copyItem, pasteItem);
95     // Bug or restriction--must add to context menu first
96     // http://bugs.java.com/bugdatabase/view_bug.do?bug_id=JDK-8194270
97
98     // demonstrate nested menus
99
100    Menu optionsMenu = new Menu("Options",
101        new ImageView("menu/options.gif"), readOnlyItem,
102        insertItem, oertypeItem);
103
104    editMenu.getItems().add(optionsMenu);
105
106    // demonstrate mnemonics
107
108    MenuItem aboutProgramItem = new MenuItem("_About this program");
109    MenuItem aboutCoreJavaItem = new MenuItem("About _Core Java");
110    Menu helpMenu = new Menu("_Help", null,
111        aboutProgramItem, aboutCoreJavaItem);
112
113    // add menu bar
114
115    MenuBar bar = new MenuBar(fileMenu, editMenu, helpMenu);
116    VBox root = new VBox(bar, textArea);
117    for (Menu menu : bar.getMenus()) defaultAction(menu, event ->
118        {
119            MenuItem item = (MenuItem) event.getSource();
120            textArea.appendText(item.getText() + " selected\n");
121        });
122
123    stage.setScene(new Scene(root));
124    stage.setTitle("MenuTest");
125    stage.show();
126 }
127 }
```

javafx.scene.control.MenuItem

- `MenuItem(String text)`
- `MenuItem(String text, Node graphic)`
constructs a menu item with the given text and graphic.
- `public void setOnAction(EventHandler<ActionEvent> value)`
sets the handler that is called when this item is selected.
- `Menu getParentMenu()`
returns the menu in which this item is contained.
- `void setAccelerator(KeyCombination value)`
sets the accelerator for this menu item.

javafx.scene.input.KeyCombination

- `static KeyCombination keyCombination(String name)`
yields a key combination for the given description. Use modifiers (Shift, Ctrl, Alt, Meta, Shortcut) followed by +, then the key name, as produced by the `KeyCode.getName` method; for example, "Ctrl+Shift+F11".

javafx.scene.control.Menu

- `Menu(String text)`
- `Menu(String text, Node graphic)`
- `Menu(String text, Node graphic, MenuItem... items)`
constructs a menu with the given text, graphic, and items.
- `ObservableList<MenuItem> getItems()`
returns the mutable list of items in this menu.

javafx.scene.control.MenuBar

- `MenuBar(Menu... menus)`
constructs a menu bar with the given menus.
- `ObservableList<Menu> getMenus()`
returns the mutable list of menus in this menu bar.

javafx.scene.control.CheckMenuItem

- `CheckMenuItem(String text)`
constructs a check menu item with the given text and graphic.
- `CheckMenuItem(String text, Node graphic)`
constructs a check menu item with the given text and graphic.
- `boolean isSelected()`
returns true if this item is selected.

javafx.scene.control.RadioMenuItem

- `RadioMenuItem(String text)`
- `RadioMenuItem(String text, Node graphic)`
constructs a radio menu item with the given text and graphic.
- `boolean isSelected()`
returns true if this item is selected.
- `void setToggleGroup(ToggleGroup value)`
sets the toggle group for this item.

javafx.scene.control.ContextMenu

- `ContextMenu(MenuItem... items)`
constructs a context menu with the given items.

javafx.scene.control.Control

- `void setContextMenu(ContextMenu value)`
sets the context menu for this control.

javafx.scene.Node

- `void setOnContextMenuRequested(EventHandler<? super ContextMenuEvent> value)`
sets the handler for right-clicking on this node.

13.5.4 Simple Dialogs

JavaFX has a set of ready-made simple dialogs for displaying messages and for obtaining a single input.

To show a message, construct an `Alert` object by supplying the alert type and text. Then call the `showAndWait` method.

```
Alert alert = new Alert(Alert.AlertType.INFORMATION,  
    "Everything is fine.");  
alert.showAndWait();
```

The dialog is displayed until the user dismisses it (by clicking the OK button, closing the window, or hitting the Escape key). By changing the type to the `CONFIRMATION`, `WARNING`, or `ERROR` type, the header text and image change—see Figure 13.24.

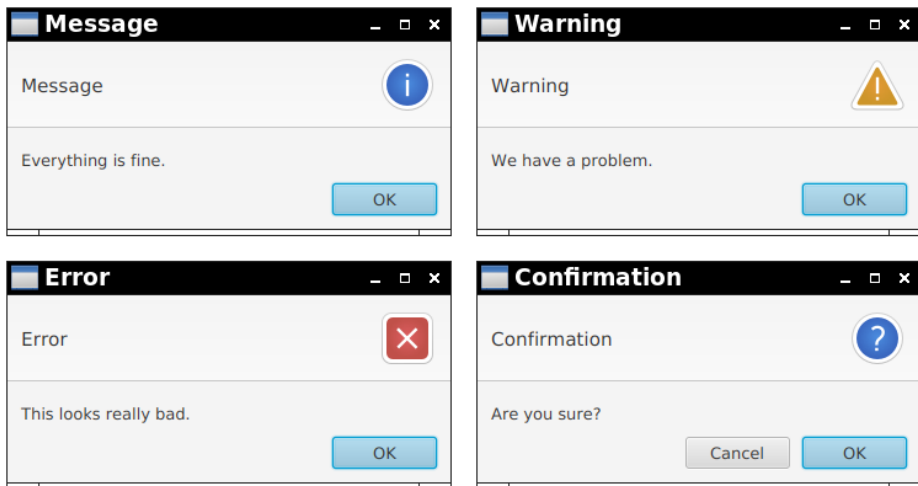


Figure 13.24 Alert dialogs

The `CONFIRMATION` alert type shows two buttons. To see whether the user confirms or cancels, you need to capture the return value of the `showAndWait` method. It has type `Optional<ButtonType>`. The `ButtonType` enumeration lists the button types that are allowed in an alert dialog. The `Optional` class, which we will discuss in Chapter 1 of Volume II, denotes a value that may be present or absent. A confirmation dialog always returns `ButtonType.OK` or `ButtonType.CANCEL`, wrapped in an `Optional`. Since there is no possibility of an empty optional, you can call the `get` method to find out whether the user has clicked OK:

```
if (alert.showAndWait().get() == ButtonType.OK)  
{  
    // the user has confirmed  
    . . .  
}
```

You can add any of the supported button types into an alert—see Figure 13.25. List the button types in the constructor:

```
Alert alert = new Alert(Alert.AlertType.NONE,
    "Now what?",
    ButtonType.NEXT, ButtonType.PREVIOUS, ButtonType.FINISH);
```

The `showAndWait` method returns the selected button type, wrapped in an `Optional`, which is never empty.



NOTE: If a dialog box has more than one button and no Cancel button, it cannot be closed. The user must click on one of the buttons in order to proceed.

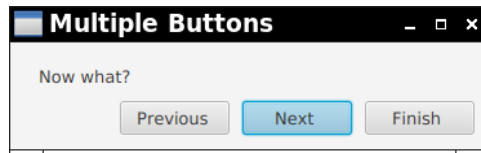


Figure 13.25 An alert dialog with multiple buttons

The alert dialog in Figure 13.25 has alert type `NONE`. Since the header text and graphic are null for this type, no header is shown.

You can supply your own header text and graphic, as we do in the example shown in Figure 13.26:

```
alert.setHeaderText("Exception");
alert.setGraphic(new ImageView("dialogs/bomb.png"));
```

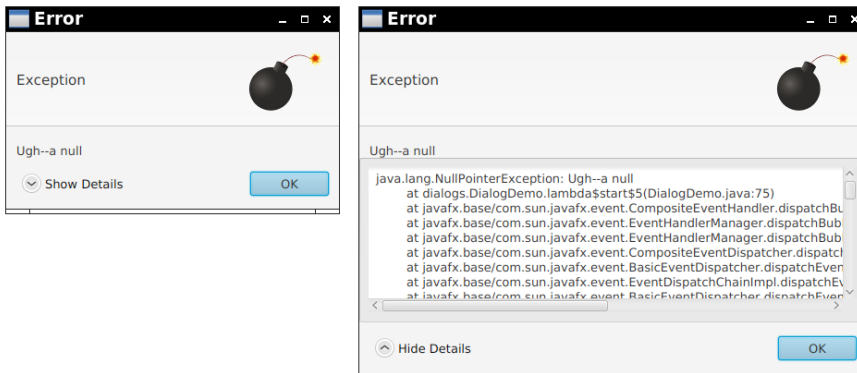


Figure 13.26 Expandable content

This dialog also shows an *expandable content* node that is revealed by clicking on “Show Details”—in our case, a text area containing an exception stack trace. Here is how you set such a node:

```
TextArea stackTrace = new TextArea();  
alert.getDialogPane().setExpandableContent(stackTrace);
```

If you want to accept text input from the user, use a `TextInputDialog`, shown in Figure 13.27.



Figure 13.27 A text input dialog

The `showAndWait` method returns an `Optional<String>`, which is empty if the user clicked the Cancel button or dismissed the dialog. Here is one way of processing it:

```
TextInputDialog dialog = new TextInputDialog();  
dialog.setHeaderText("What is your name?");  
dialog.showAndWait().ifPresentOrElse(  
    result -> { do something with result },  
    () -> { deal with cancellation }  
);
```

The `ChoiceDialog` class is similar. You provide a default choice and an array or list of choices. If the default is not null, it is selected when the dialog is shown (see Figure 13.28). The `showAndWait` method returns an `Optional` that contains the selected item, or is empty if the dialog was canceled.

```
ChoiceDialog<String> dialog = new ChoiceDialog<>("System", Font.getFamilies());  
dialog.setHeaderText("Pick a font.");  
Optional<String> choice = dialog.showAndWait();
```

The type parameter of the generic `ChoiceDialog<T>` class is the type of the choices. You can provide choices of any type, provided that their `toString` method yields meaningful strings for the combo box.

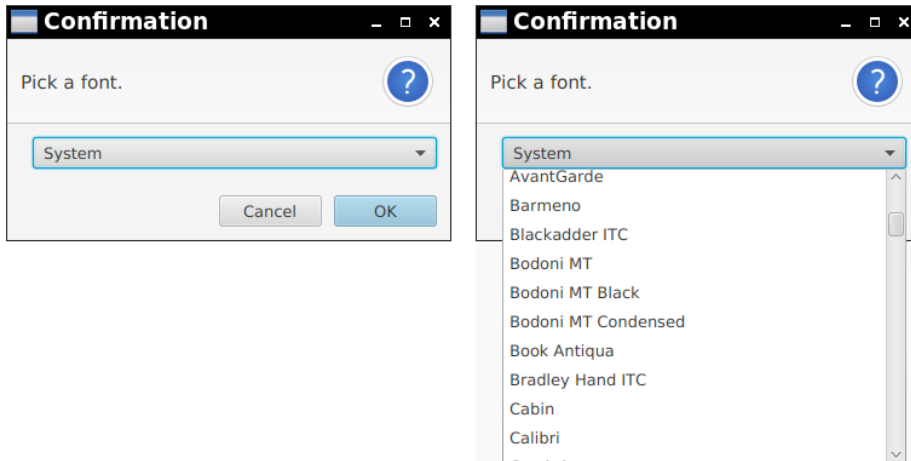


Figure 13.28 A choice dialog



CAUTION: Unlike the `ComboBox<T>` control, the `ChoiceDialog<T>` control does not allow you to set a string converter.

Finally, there is a `FileChooser` that allows users to pick one or more files. The dialog is native to the system and does not look like the other JavaFX dialogs. You can specify the default directory as well as filters for file extensions.

```
FileChooser dialog = new FileChooser();
dialog.setInitialDirectory(new File("images"));
dialog.getExtensionFilters().addAll(
    new FileChooser.ExtensionFilter("GIF images", "*.gif"),
    new FileChooser.ExtensionFilter("JPEG images", "*.jpg"));
File result = dialog.showSaveDialog(stage);
```

When showing the dialog, provide the `Stage` object, as shown. If you pass `null`, then input to the stage is not blocked.

The `showSaveDialog` method produces a dialog that lets you pick an existing file or type in a new name. The `showOpenDialog` method only lets you pick existing files. There is also a `showOpenMultipleDialog` method that returns a `List` of all selected files. If the dialog is canceled, all three methods return `null`.

To pick a directory, use a `DirectoryChooser` instead:

```
DirectoryChooser dialog = new DirectoryChooser();
File result = dialog.showDialog(stage);
```

Listing 13.16 lets you experiment with all the dialogs that were introduced in this section.

Listing 13.16 dialogs/DialogDemo.java

```
1 package dialogs;
2
3 import java.io.*;
4
5 import javafx.application.*;
6 import javafx.geometry.*;
7 import javafx.scene.*;
8 import javafx.scene.control.*;
9 import javafx.scene.image.*;
10 import javafx.scene.layout.*;
11 import javafx.scene.text.*;
12 import javafx.stage.*;
13
14 /**
15  * @version 1.0 2017-12-29
16  * @author Cay Horstmann
17  */
18 public class DialogDemo extends Application
19 {
20     public void start(Stage stage)
21     {
22         TextArea textArea = new TextArea();
23
24         Button information = new Button("Information");
25         information.setOnAction(event ->
26         {
27             Alert alert = new Alert(Alert.AlertType.INFORMATION,
28                 "Everything is fine.");
29             alert.showAndWait();
30         });
31
32         Button warning = new Button("Warning");
33         warning.setOnAction(event ->
34         {
35             Alert alert = new Alert(Alert.AlertType.WARNING,
36                 "We have a problem.");
37             alert.showAndWait();
38         });
39
40         Button error = new Button("Error");
```



```
41 error.setOnAction(event ->
42     {
43         Alert alert = new Alert(Alert.AlertType.ERROR,
44             "This looks really bad.");
45         alert.showAndWait();
46     });
47
48 Button confirmation = new Button("Confirmation");
49 confirmation.setOnAction(event ->
50     {
51         Alert alert = new Alert(Alert.AlertType.CONFIRMATION,
52             "Are you sure?");
53         if (alert.showAndWait().get() == ButtonType.OK)
54             textArea.appendText("Confirmed\n");
55         else
56             textArea.appendText("Canceled\n");
57     });
58
59 Button multipleButtons = new Button("Multiple Buttons");
60 multipleButtons.setOnAction(event ->
61     {
62         Alert alert = new Alert(Alert.AlertType.NONE,
63             "Now what?",
64             ButtonType.NEXT, ButtonType.PREVIOUS, ButtonType.FINISH);
65         alert.setTitle("Multiple Buttons");
66         textArea.appendText(alert.showAndWait() + "\n");
67     });
68
69 Button expandableContent = new Button("Expandable Content");
70 expandableContent.setOnAction(event ->
71     {
72         Throwable t = new NullPointerException("Ugh--a null");
73         Alert alert = new Alert(Alert.AlertType.ERROR,
74             t.getMessage());
75         alert.setHeaderText("Exception");
76         alert.setGraphic(new ImageView("dialogs/bomb.png"));
77
78         TextArea stackTrace = new TextArea();
79         StringWriter out = new StringWriter();
80         t.printStackTrace(new PrintWriter(out));
81         stackTrace.setText(out.toString());
82         alert.getDialogPane().setExpandableContent(stackTrace);
83
84         textArea.appendText(alert.showAndWait() + "\n");
85     });
86
87 Button textInput = new Button("Text input");
```

(Continues)

Listing 13.16 *(Continued)*

```

88     textInput.setOnAction(event ->
89     {
90         TextInputDialog dialog = new TextInputDialog();
91         dialog.setHeaderText("What is your name?");
92         dialog.showAndWait().ifPresentOrElse(
93             result -> textArea.appendText("Name: " + result + "\n"),
94             () -> textArea.appendText("Canceled\n"));
95     });
96
97     Button choiceDialog = new Button("Choice Dialog");
98     choiceDialog.setOnAction(event ->
99     {
100         ChoiceDialog<String> dialog = new ChoiceDialog<>("System",
101             Font.getFamilies());
102         dialog.setHeaderText("Pick a font.");
103         dialog.showAndWait().ifPresentOrElse(
104             result -> textArea.appendText("Selected: " + result + "\n"),
105             () -> textArea.appendText("Canceled\n"));
106     });
107
108     Button fileChooser = new Button("File Chooser");
109     fileChooser.setOnAction(event ->
110     {
111         FileChooser dialog = new FileChooser();
112         dialog.setInitialDirectory(new File("menu"));
113         dialog.setInitialFileName("untitled.gif");
114         dialog.getExtensionFilters().addAll(
115             new FileChooser.ExtensionFilter("GIF images", "*.gif"),
116             new FileChooser.ExtensionFilter("JPEG images", "*.jpg", "*.jpeg"));
117         File result = dialog.showSaveDialog(stage);
118         if (result == null)
119             textArea.appendText("Canceled\n");
120         else
121             textArea.appendText("Selected: " + result + "\n");
122     });
123
124     Button directoryChooser = new Button("Directory Chooser");
125     directoryChooser.setOnAction(event ->
126     {
127         DirectoryChooser dialog = new DirectoryChooser();
128         File result = dialog.showDialog(stage);
129         if (result == null)
130             textArea.appendText("Canceled\n");
131         else
132             textArea.appendText("Selected: " + result + "\n");
133     });
134
135     final double rem = new Text("").getLayoutBounds().getHeight();

```

```

136     VBox buttons = new VBox(0.8 * rem,
137         information, warning, error, confirmation,
138         multipleButtons, expandableContent, textInput, choiceDialog,
139         fileChooser, directoryChooser);
140     buttons.setPadding(new Insets(0.8 * rem));
141
142     HBox root = new HBox(textArea, buttons);
143
144     stage.setScene(new Scene(root));
145     stage.setTitle("DialogDemo");
146     stage.show();
147 }
148 }

```

javafx.scene.control.Alert

- `Alert(Alert.AlertType alertType)`
constructs an alert dialog of the given type. The `Alert.AlertType` enumeration has values `INFORMATION`, `WARNING`, `ERROR`, `CONFIRMATION`, and `NONE`.
- `Alert(Alert.AlertType alertType, String contentText, ButtonType... buttons)`
constructs an alert dialog of the given type with the given message and buttons. The `ButtonType` enumeration has values `OK`, `CANCEL`, `YES`, `NO`, `NEXT`, `PREVIOUS`, `FINISH`, `APPLY`, and `CLOSE`. Both `CANCEL` and `CLOSE` cancel the dialog.

javafx.scene.control.Dialog<T>

- `Optional<T> showAndWait()`
shows the dialog and waits for user input. Returns an object of type `T` representing the user input, wrapped in an `Optional`, or an empty `Optional` if the dialog was dismissed.
- `void setHeaderText(String value)`
sets the text that is displayed in the header of this dialog.
- `void setGraphic(Node value)`
sets the graphic that is displayed in this dialog.
- `DialogPane getDialogPane()`
gets the pane containing all controls of this dialog.

javafx.scene.control.DialogPane

- `void setExpandableContent(Node content)`
sets a node that can be expanded and hidden.

javafx.scene.control.TextInputDialog

- `TextInputDialog()`
constructs a dialog for entering a string.

javafx.scene.control.ChoiceDialog

- `ChoiceDialog(T defaultChoice, T... choices)`
- `ChoiceDialog(T defaultChoice, Collection<T> choices)`
constructs a dialog for picking an item of type `T`. If `defaultChoice` is not null, it is selected.

javafx.stage.FileChooser

- `FileChooser()`
constructs a file chooser.
- `File showOpenDialog(Window ownerWindow)`
- `List<File> showOpenMultipleDialog(Window ownerWindow)`
returns the chosen file or null if the dialog was dismissed. The `ownerWindow` is blocked from receiving input while the dialog is displayed.
- `File showSaveDialog(Window ownerWindow)`
shows a dialog for selecting an existing file or typing a new file and returns the chosen file or null if the dialog was dismissed. The `ownerWindow` is blocked from receiving input while the dialog is displayed.
- `void setInitialDirectory(File value)`
sets the initial directory for this file chooser.

FileChooser.ExtensionFilter

- `ExtensionFilter(String description, String... extensions)`
- `ExtensionFilter(String description, List<String> extensions)`
constructs an extension filter that accepts files with any of the given extensions. The extension strings have the format `*.extension`.

javafx.stage.DirectoryChooser

- `DirectoryChooser()`
constructs a directory chooser.
- `File showDialog(Window ownerWindow)`
shows a dialog for selecting an existing directory or creating a new one, and returns the chosen directory or null if the dialog was dismissed. The `ownerWindow` is blocked from receiving input while the dialog is displayed.
- `void setInitialDirectory(File value)`
sets the initial directory for this file chooser.

13.5.5 Fancy Controls

Of course, JavaFX has tab panes, trees, and tables, just like Swing does, as well as a few user interface controls that Swing never got, such as a date picker and an accordion. In this section, I want to dispel any remaining Swing nostalgia by showing you three fancy controls that are far beyond anything Swing had to offer.

Figure 13.29 shows one of many charts that you can make with JavaFX, out of the box, without having to install any third-party libraries.

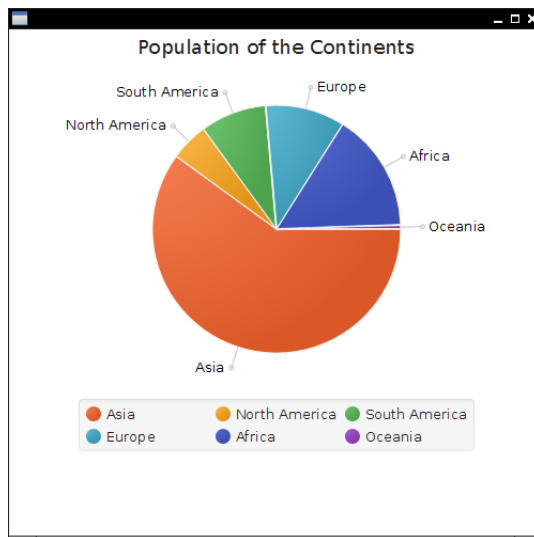


Figure 13.29 A JavaFX pie chart

And it's easy as pie:

```
PieChart chart = new PieChart();
chart.getData().addAll(
    new PieChart.Data("Asia", 4298723000.0),
    new PieChart.Data("North America", 355361000.0),
    new PieChart.Data("South America", 616644000.0),
    new PieChart.Data("Europe", 742452000.0),
    new PieChart.Data("Africa", 1110635000.0),
    new PieChart.Data("Oceania", 38304000.0));
chart.setTitle("Population of the Continents");
```

Altogether, there are half a dozen chart types that you can use and customize. See <https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/charts.htm> for more information.

In Swing, you could show HTML in a `JEditorPane`, but the rendering was poor for most real-world HTML. That's understandable—implementing a browser is hard work. In fact, it is so hard that most browsers are built on top of the open source WebKit engine. JavaFX does the same. A `WebView` displays an embedded native WebKit window (see Figure 13.30).

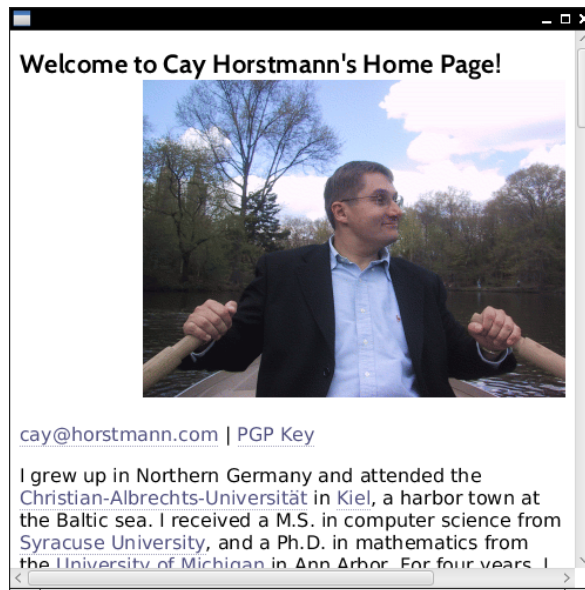


Figure 13.30 Browsing the Web (Photo source: Cay Horstmann)

Here is the code to show a web page:

```
String location = "http://horstmann.com";
WebView browser = new WebView();
WebEngine engine = browser.getEngine();
engine.load(location);
```

The browser is live—you can click on links in the usual way. JavaScript works as well. However, if you want to display status line or popup messages from JavaScript, you need to install notification handlers and implement your own status line and popups.



NOTE: `WebView` does not support any plugins, so you cannot use it to show Flash animations or PDF documents. It also doesn't show applets.

Prior to JavaFX, media playback was pitiful in Java. A Java Media Framework was available as an optional download, but it did not get much love from the developers. Of course, implementing audio and video playback is even harder than writing a browser. Therefore, JavaFX leverages an existing toolkit: the open source GStreamer framework.

To play a video, construct a `Media` object from a URL string, construct a `MediaPlayer` to play it, and use a `MediaView` to show the player:

```
Path path = Paths.get("moonlanding.mp4");
Media media = new Media(path.toUri().toString());
MediaPlayer player = new MediaPlayer(media);
player.setAutoplay(true);
MediaView video = new MediaView(player);
video.setOnError(System.out::println);
```

As you can see in Figure 13.31, the video is played—but, unfortunately, there are no video controls. You can add your own (see <https://docs.oracle.com/javase/8/javafx/media-tutorial/playercontrol.htm>), but it would have been nice to supply a default set of controls.



NOTE: Ever so often, GStreamer can't handle a particular video file. The error handler in the code sample displays GStreamer messages so that you can diagnose playback problems.

Listing 13.17 shows the complete code for all examples in this section.



Figure 13.31 Playing a video (Image source: NASA)

Listing 13.17 fancy/FancyControls.java

```
1 package fancy;
2
3 import java.nio.file.*;
4 import javafx.application.*;
5 import javafx.geometry.*;
6 import javafx.scene.*;
7 import javafx.scene.chart.*;
8 import javafx.scene.layout.*;
9 import javafx.scene.media.*;
10 import javafx.scene.web.*;
11 import javafx.stage.*;
12
13 /**
14  * @version 1.0 2017-12-29
15  * @author Cay Horstmann
16  */
```



```
17 public class FancyControls extends Application
18 {
19     public void start(Stage stage)
20     {
21         PieChart chart = new PieChart();
22         chart.getData().addAll(
23             new PieChart.Data("Asia", 4298723000.0),
24             new PieChart.Data("North America", 355361000.0),
25             new PieChart.Data("South America", 616644000.0),
26             new PieChart.Data("Europe", 742452000.0),
27             new PieChart.Data("Africa", 1110635000.0),
28             new PieChart.Data("Oceania", 38304000.0));
29         chart.setTitle("Population of the Continents");
30
31         String location = "http://horstmann.com";
32         WebView browser = new WebView();
33         WebEngine engine = browser.getEngine();
34         engine.load(location);
35
36         Path path = Paths.get("fancy/moonlanding.mp4");
37         Media media = new Media(path.toUri().toString());
38         MediaPlayer player = new MediaPlayer(media);
39         player.setAutoplay(true);
40         MediaView video = new MediaView(player);
41         video.setOnError(ex -> System.out.println(ex));
42
43         stage.setWidth(500);
44         stage.setHeight(500);
45         stage.setScene(new Scene(browser));
46         stage.show();
47
48         Stage stage2 = new Stage();
49         stage2.setWidth(500);
50         stage2.setHeight(500);
51         stage2.setX(stage.getX() + stage.getWidth());
52         stage2.setY(stage.getY());
53         stage2.setScene(new Scene(chart));
54         stage2.show();
55
56         HBox box = new HBox(video);
57         box.setAlignment(Pos.CENTER);
58         Stage stage3 = new Stage();
59         stage3.setWidth(500);
60         stage3.setHeight(500);
61         stage3.setX(stage.getX());
62         stage3.setY(stage.getY() + stage.getHeight());
63         stage3.setScene(new Scene(box));
64         stage3.show();
65     }
66 }
```

13.6 Properties and Bindings

A *property* is an attribute of a class that you can read or write. Commonly, the property is backed by a field, and the property getter and setter simply read and write that field. But the getter and setter can also take other actions, such as reading values from a database or sending out change notifications.

In JavaFX, properties are particularly important because it is easy to “bind” them so that one property is updated when another property changes. In the following sections, we will discuss properties and bindings in detail.

13.6.1 JavaFX Properties

In many programming languages, there is convenient syntax for invoking property getters and setters. Using the property on the right-hand side of an assignment calls the getter, and using it on the left-hand side calls the setter.

```
value = obj.property; // in many languages (but not Java), this calls the property getter
obj.property = value; // and this calls the property setter
```

Sadly, Java does not have such syntax. But it has supported properties by convention since Java 1.1. The JavaBeans specification states that a property should be inferred from a getter/setter pair. For example, a class with methods `String getText()` and `void setText(String newValue)` is deemed to have a text property. The `Introspector` and `BeanInfo` classes in the `java.beans` package let you enumerate all properties of a class.

The JavaBeans specification also defines *bound properties*, where objects emit property change events when setters are invoked. JavaFX does not make use of this part of the specification. Instead, a JavaFX property has a third method, besides the getter and setter, that returns an object implementing the `Property` interface. For example, a JavaFX text property has three methods

```
String getText()
void setText(String value)
Property<String> textProperty()
```



NOTE: The `Node` class has over 80 JavaFX properties, and subclasses such as `Rectangle` or `Button` have well over a hundred. The Java API documentation lists them separately, before the constructors and methods.

You can attach a listener to the property object. That’s different from old-fashioned JavaBeans. In JavaFX, the property object, not the bean, sends out notifications. There is a good reason for this change. Implementing bound

JavaBeans properties required boilerplate code to add, remove, and fire listeners; in JavaFX it's much simpler because there are library classes that do much of that work.

Let's see how we can implement a property text in a class `Greeting`. Here is the simplest way to do that:

```
public class Greeting
{
    private StringProperty text = new SimpleStringProperty("");

    public final StringProperty textProperty() { return text; }
    public final void setText(String newValue) { text.set(newValue); }
    public final String getText() { return text.get(); }
}
```

The `StringProperty` class wraps a string. It has methods for getting and setting the wrapped value and for managing listeners.

As you can see, implementing a JavaFX property requires some boilerplate code, and there is unfortunately no way in Java to generate the code automatically. But at least you won't have to write the code for managing listeners.

It is not a requirement to declare property getters and setters as `final`, but the JavaFX designers recommend it.



NOTE: With this pattern, a property object is needed for each property, whether anyone listens to it or not. If you implement a class with many properties, and you expect that many instances of that class will be constructed, you should instantiate the property objects on demand. Use a regular field for holding the property value, and switch to a property object only when someone calls the `xxxProperty()` method.

In the preceding example, we defined a `StringProperty`. For a primitive-type property, use one of `IntegerProperty`, `LongProperty`, `DoubleProperty`, `FloatProperty`, or `BooleanProperty`. There are also `ListProperty`, `MapProperty`, and `SetProperty` classes. For everything else, use an `ObjectProperty<T>`. All these are abstract classes with concrete subclasses `SimpleIntegerProperty`, `SimpleObjectProperty<T>`, and so on.



NOTE: If all you care about is managing listeners and bindings, your property methods can return objects of type `ObjectProperty<T>`, or even the `Property<T>` interface. The more specialized classes are useful to make computations with the properties, as explained in the next section.

There are two kinds of listeners that can be attached to a property. A `ChangeListener` is notified when the property value has changed, and an `InvalidationListener` is called when the property value *may* have changed. The distinction matters for a property with lazy evaluation. As you will see in the next section, some properties are computed from others, and the computation is only done when necessary. The `ChangeListener` callback tells you the old and new values, which means it has to compute the new value. The `InvalidationListener` doesn't compute the new value, but that means you might get a callback when the value hasn't actually changed. In most situations, the difference between the two is immaterial.

Here is how to add an invalidation listener that is called whenever the text property of a greeting changes:

```
Greeting greeting = new Greeting();
greeting.setText("Hello");
greeting.textProperty().addListener(event ->
{
    System.out.println("greeting is now " + greeting.getText());
});
greeting.setText("Goodbye"); // the listener is now invoked
```

In contrast, here is how you attach a change listener:

```
greeting.textProperty().addListener((property, oldValue, newValue) ->
{
    System.out.println("greeting is now " + newValue);
});
```



CAUTION: It is a bit tricky to use the `ChangeListener` interface for numeric properties. One would like to call

```
slider.valueProperty().addListener((property, oldValue, newValue) ->
    message.setFont(Font.font(newValue)));
```

But that does not work. `DoubleProperty` implements `Property<Number>` and not `Property<Double>`. Therefore, the type for `oldValue` and `newValue` is `Number` and not `Double`, so you have to unbox manually:

```
slider.valueProperty().addListener((property, oldValue, newValue) ->
    message.setFont(Font.font(newValue.doubleValue())));
```

13.6.2 Bindings

The *raison d'être* for JavaFX properties is the notion of *binding*: automatically updating one property when another one changes. Consider, for example,

the application in Figure 13.32. When the user edits the shipping address at the top, the billing address at the bottom updates, too.

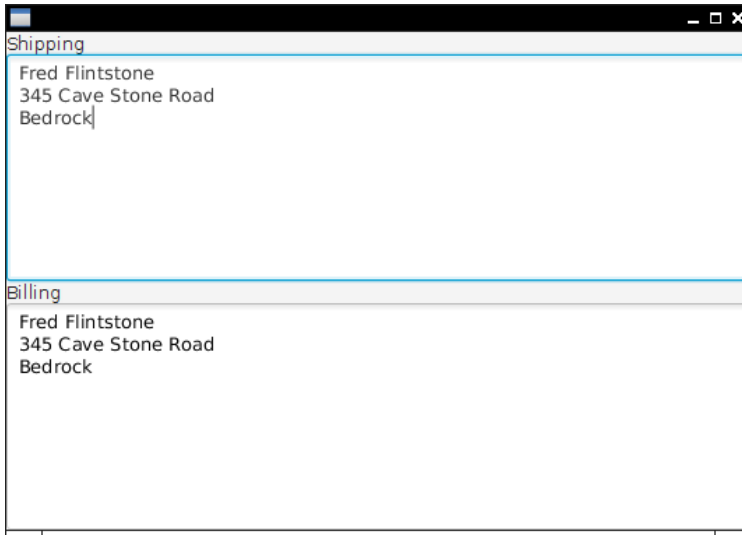


Figure 13.32 The bound text property updates automatically.

This is achieved by binding one property to the other:

```
billing.textProperty().bind(shipping.textProperty());
```

Under the hood, a change listener is added to the text property of shipping that sets the text property of billing.

You can also call

```
billing.textProperty().bindBidirectional(shipping.textProperty());
```

Now, when either of these properties changes, the other is updated.

To undo a binding, call `unbind` or `unbindBidirectional`.

The binding mechanism solves a common problem in user interface programming. For example, consider a date field and a calendar picker. When the user picks a date from the calendar, the date field should automatically update, as should be the date property of the model.

Of course, in many situations, one property depends on another, but the relationship is more complex. Consider the program shown in Figure 13.33. We always want the circle centered in the scene. That is, its `centerX` property should be one half of the `width` property of the scene.

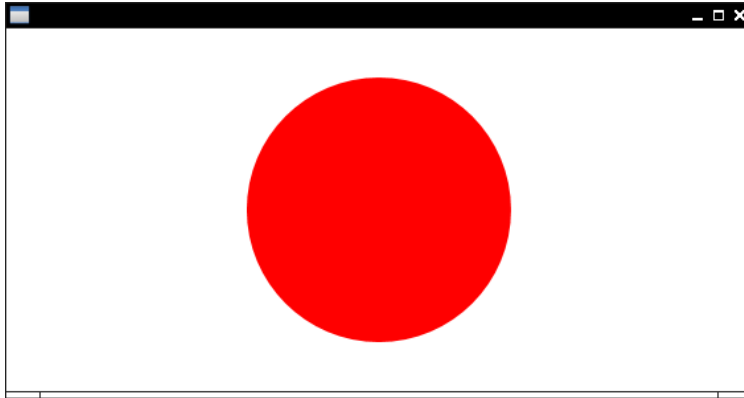


Figure 13.33 The center of this circle is bound to half the width and height of the scene.

To achieve this, we need to produce a computed property. The `Bindings` class has static methods for this purpose. For example, `Bindings.divide(scene.widthProperty(), 2)` is a property whose value is one half of the scene width. When the scene width changes, so does that property. All that remains is to bind that computed property to the circle's `centerX` property:

```
circle.centerXProperty().bind(Bindings.divide(scene.widthProperty(), 2));
```



NOTE: Alternatively, you can call `scene.widthProperty().divide(2)`. With more complex expressions, the static `Bindings` methods seems a bit easier to read, particularly if you use

```
import static javafx.beans.binding.Bindings.*;

and write divide(scene.widthProperty(), 2).
```

Here is a more realistic example. We want to disable the **Smaller** and **Larger** buttons when the gauge is too small or large (Figure 13.34).

```
smaller.disableProperty().bind(Bindings.lessThanOrEqual(gauge.widthProperty(), 0));
larger.disableProperty().bind(Bindings.greaterThanOrEqual(gauge.widthProperty(), 100));
```

When the width is ≤ 0 , the **Smaller** button is disabled. When the width is ≥ 100 , the **Larger** button is disabled.

Table 13.3 lists all operators that the `Bindings` class provides. One or both of the arguments implement the `Observable` interface or one of its subinterfaces. The `Observable` interface provides methods for adding and removing an `InvalidationListener`. The `ObservableValue` interface adds `ChangeListener` management

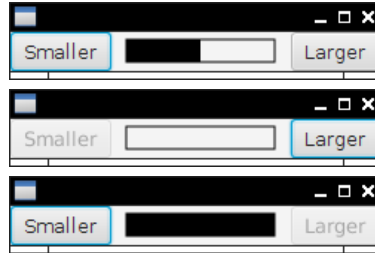


Figure 13.34 When the gauge reaches either end, a button is disabled.

and a `getValue` method. Its subinterfaces provide methods to get the value in the appropriate type. For example, the `get` method of `ObservableStringValue` returns a `String` and the `get` method of `ObservableIntegerValue` returns an `int`. The return types of the methods of the Bindings are subinterfaces of the `Binding` interface, itself a subinterface of the `Observable` interface. A `Binding` knows about all properties on which it depends.

In practice, you don't need to worry about all of these interfaces. You combine properties and you get something that you can bind to another property.

Table 13.3 Operators Supplied by the Bindings Class

Method Name	Arguments
add, subtract, multiply, divide, max, min	An <code>ObservableNumberValue</code> (as first or second argument) and an <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> , or another <code>ObservableNumberValue</code> .
negate	An <code>ObservableNumberValue</code> .
greaterThan, greaterThanOrEqualTo, lessThan, lessThanOrEqualTo	An <code>ObservableNumberValue</code> and an <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> , or <code>ObservableNumberValue</code> ; or an <code>ObservableStringValue</code> and a <code>String</code> or <code>ObservableStringValue</code> .
equal, notEqual	An <code>ObservableNumberValue</code> and an <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> , or <code>ObservableNumberValue</code> ; or an <code>ObservableStringValue</code> and a <code>String</code> or <code>ObservableStringValue</code> ; or an <code>ObservableObjectValue</code> and a <code>Object</code> or <code>ObservableObjectValue</code> .
equalIgnoreCase, notEqualIgnoreCase	An <code>ObservableStringValue</code> and a <code>String</code> or <code>ObservableStringValue</code> .
isEmpty, isEmpty	An <code>Observable(List Map Set StringValue)</code> .

(Continues)

Table 13.3 (Continued)

Method Name	Arguments
<code>isNull, isNotNull</code>	An <code>ObservableObjectValue</code> .
<code>length</code>	An <code>ObservableStringValue</code> .
<code>size</code>	An <code>Observable(List Map Set)</code> .
<code>and, or</code>	Two <code>ObservableBooleanValue</code> .
<code>not</code>	An <code>ObservableBooleanValue</code> .
<code>convert</code>	An <code>ObservableValue</code> that is converted to a string binding.
<code>concat</code>	A sequence of objects whose <code>toString</code> values are concatenated. If any of the objects is an <code>ObservableValue</code> that changes, the concatenation changes too.
<code>format</code>	An optional locale, a <code>MessageFormat</code> string, and a sequence of objects that are formatted. If any of the objects is an <code>ObservableValue</code> that changes, the formatted string changes too.
<code>valueAt (double float integer long)ValueAt stringValueAt</code>	An <code>ObservableList</code> and an index, or an <code>ObservableMap</code> and a key.
<code>create(Boolean Double Float Integer Long Object String)Binding</code>	A <code>Callable</code> and a list of dependencies.
<code>select select(Boolean Double Float Integer Long String)</code>	An <code>Object</code> or <code>ObservableValue</code> and a sequence of public property names, yielding the property <i>obj.p₁.p₂. . . .p_n</i> .
<code>when</code>	Yields a builder for a conditional operator. The binding <code>when(b).then(v₁).otherwise(v₂)</code> yields <i>v₁</i> or <i>v₂</i> , depending on whether the <code>ObservableBooleanValue</code> <i>b</i> is true or not. Here, <i>v₁</i> or <i>v₂</i> can be regular or observable values. The conditional value is recomputed whenever an observable value changes.

Building up a computed property with the methods of the `Bindings` class can get quite baroque. There is another approach for producing computed bindings that you may find easier. Simply put the expression that you want to have

computed into a lambda, and supply a list of dependent properties. When any of the properties changes, the lambda is recomputed. For example,

```
larger.disableProperty().bind(
    createBooleanBinding(
        () -> gauge.getWidth() >= 100, // this expression is computed. . .
        gauge.widthProperty())); // . . . when this property changes
```



NOTE: In the JavaFX Script language, the compiler analyzed binding expressions and automatically figured out the dependent properties. You just declared `larger.disable bind gauge.width >= 100`, and the compiler attached a listener to the `gauge.width` property. Of course, in Java, the programmer needs to supply this information.

Listings 13.18 to 13.20 show the complete programs for the three examples in this section.

Listing 13.18 binding/BindingDemo1.java

```
1 package binding;
2
3 import javafx.application.*;
4 import javafx.scene.*;
5 import javafx.scene.control.*;
6 import javafx.scene.layout.*;
7 import javafx.stage.*;
8
9 /**
10  @version 1.0 2017-12-29
11  @author Cay Horstmann
12  */
13 public class BindingDemo1 extends Application
14 {
15     public void start(Stage stage)
16     {
17         TextArea shipping = new TextArea();
18         TextArea billing = new TextArea();
19         billing.textProperty().bindBidirectional(shipping.textProperty());
20         VBox root = new VBox(
21             new Label("Shipping"), shipping,
22             new Label("Billing"), billing);
23         Scene scene = new Scene(root);
24         stage.setScene(scene);
25         stage.show();
26     }
27 }
```

Listing 13.19 binding/BindingDemo2.java

```
1 package binding;
2
3 import javafx.application.*;
4 import javafx.beans.binding.*;
5 import javafx.scene.*;
6 import javafx.scene.layout.*;
7 import javafx.scene.paint.*;
8 import javafx.scene.shape.*;
9 import javafx.stage.*;
10
11 /**
12  * @version 1.0 2017-12-29
13  * @author Cay Horstmann
14  */
15 public class BindingDemo2 extends Application
16 {
17     public void start(Stage stage)
18     {
19         Circle circle = new Circle(100, 100, 100);
20         circle.setFill(Color.RED);
21         Pane pane = new Pane(circle);
22         Scene scene = new Scene(pane);
23         circle.centerXProperty().bind(
24             Bindings.divide(scene.widthProperty(), 2));
25         circle.centerYProperty().bind(
26             Bindings.divide(scene.heightProperty(), 2));
27         stage.setScene(scene);
28         stage.show();
29     }
30 }
```

Listing 13.20 binding/BindingDemo3.java

```
1 package binding;
2
3 import static javafx.beans.binding.Bindings.*;
4
5 import javafx.application.*;
6 import javafx.scene.*;
7 import javafx.scene.control.*;
8 import javafx.scene.layout.*;
9 import javafx.scene.paint.*;
10 import javafx.scene.shape.*;
11 import javafx.stage.*;
12
```

```
13  /**
14   @version 1.0 2017-12-29
15   @author Cay Horstmann
16  */
17  public class BindingDemo3 extends Application
18  {
19      public void start(Stage stage)
20      {
21          Button smaller = new Button("Smaller");
22          Button larger = new Button("Larger");
23          Rectangle gauge = new Rectangle(0, 5, 50, 15);
24          Rectangle outline = new Rectangle(0, 5, 100, 15);
25          outline.setFill(null);
26          outline.setStroke(Color.BLACK);
27          Pane pane = new Pane(gauge, outline);
28
29          smaller.setOnAction(
30              event -> gauge.setWidth(gauge.getWidth() - 10));
31          larger.setOnAction(
32              event -> gauge.setWidth(gauge.getWidth() + 10));
33
34          // using Bindings operator
35
36          smaller.disableProperty().bind(
37              lessThanOrEqualTo(gauge.widthProperty(), 0));
38
39          // creating a binding from a lambda
40
41          larger.disableProperty().bind(
42              createBooleanBinding(
43                  () -> gauge.getWidth() >= 100, // this lambda is computed . . .
44                  gauge.widthProperty())); // . . . when this property changes
45
46          Scene scene = new Scene(new HBox(10, smaller, pane, larger));
47          stage.setScene(scene);
48          stage.show();
49      }
50  }
```

13.7 Long-Running Tasks in User Interface Callbacks

One of the reasons to use threads is to make your programs more responsive. This is particularly important in an application with a user interface. When your program needs to do something time consuming, you cannot do the work in the user-interface thread, or the user interface will be frozen. Instead, fire up another worker thread.

For example, if you want to read a file when the user clicks a button, don't do this:

```
Button open = new Button("Open");
open.setOnAction(event ->
{ // BAD--long-running action is executed on UI thread
    Scanner in = new Scanner(file);
    while (in.hasNextLine())
    {
        String line = in.nextLine();
        . . .
    }
});
```

Instead, do the work in a separate thread.

```
open.setOnAction(event ->
{ // GOOD--long-running action in separate thread
    Runnable task = () ->
    {
        Scanner in = new Scanner(file);
        while (in.hasNextLine())
        {
            String line = in.nextLine();
            . . .
        }
    };
    executor.execute(task);
});
```

However, you cannot directly update the user interface from the worker thread that executes the long-running task. User interfaces such as JavaFX, Swing, or Android are not threadsafe. You cannot manipulate user interface elements from multiple threads, or they risk becoming corrupted. In fact, JavaFX and Android check for this, and throw an exception if you try to access the user interface from a thread other than the UI thread.

Therefore, you need to schedule any UI updates to happen on the UI thread. Each user interface library provides some mechanism to schedule a `Runnable` for execution on the UI thread. For example, in JavaFX, you call

```
Platform.runLater() -> content.appendText(line + "\n");
```

It is tedious to implement user feedback in a worker thread, so user interface libraries provide some kind of helper class for managing the details, such as `SwingWorker` in Swing and `AsyncTask` in Android. You specify actions for the long-running task (which is run on a separate thread), as well as progress updates and the final disposition (which are run on the UI thread).

In JavaFX, you use the `Task<V>` class for long-running tasks. Conveniently, the class extends `FutureTask<V>`, so you don't have to learn yet another custom construct.

The `Task` class provides methods to update certain task properties in the worker thread. You bind the properties to user interface elements, which are then updated *in the UI thread*. The following properties are available:

```
String message
double progress
double workDone
double totalWork
String title
V value
```

Here we bind the `message` property to a status label:

```
status.textProperty().bind(task.messageProperty());
```

In the worker thread, call the `updateMessage` method, not `setMessage`. That method coalesces property changes. That is, if several property changes come in rapid succession, only the latest change results in a corresponding property change on the user interface thread.

```
task = new Task<>()
{
    public Integer call()
    {
        while (. . .)
        {
            . . .
            lines++;
            updateMessage(lines + " lines read");
        }
    }
};
```

To cancel a `Task`, call the `cancel` method. This will interrupt the thread executing the task. In the task, periodically call `isCanceled` to check if the task was canceled.

```
task = new Task<>()
{
    public Integer call()
    {
        while (!isCanceled() && !done)
        {
            . . .
        }
    }
};
```

Install event handlers to be notified when the task is scheduled, starts running, and when it terminates with success, due to an exception, or due to cancellation. All handlers are executed on the UI thread.

```
task.isScheduled(event -> cancel.setDisable(false));  
task.setOnRunning(event -> status.setText("Running"));  
task.setOnSucceeded(event -> status.setText("Read " + task.getValue() + " lines"));  
task.setOnFailed(event -> status.setText("Failed due to " + task.getException()));  
task.setOnCancelled(event -> status.setText("Canceled"));
```

The program in Listing 13.21 has commands for opening a text file and for canceling the file loading process. You should try the program with a long file, such as the full text of *The Count of Monte Cristo*, supplied in the gutenberg directory of the book's companion code. The file is loaded in a separate thread. While the file is being read, the Open button is disabled and the Cancel button is enabled (see Figure 13.35). After each line is read, a line counter in the status bar is updated. After the reading process is complete, the Open button is reenabled, the Cancel button is disabled, and the status line text is set to a "Done" message. We added a delay of 10 milliseconds between lines so that you can clearly observe the status updates. You would not do this in your own programs.



Figure 13.35 Reading lines from a web page in a worker thread

Pay close attention to what happens on the worker thread and the UI thread:

- The call method is executed on the worker thread.
- The lambda passed to `Platform.runLater` is executed on the UI thread.
- The handlers for the "on scheduled," "on running," "on failed," "on cancelled," and "on succeeded" events are executed on the UI thread.

- The call to `updateMessage` causes a property change on the UI thread that triggers the setter of the bound property.

Listing 13.21 `uitask/TaskDemo.java`

```
1 package uitask;
2
3 import java.io.*;
4 import java.nio.charset.*;
5 import java.util.*;
6 import java.util.concurrent.*;
7
8 import javafx.application.*;
9 import javafx.concurrent.*;
10 import javafx.geometry.*;
11 import javafx.scene.*;
12 import javafx.scene.control.*;
13 import javafx.scene.layout.*;
14 import javafx.stage.*;
15
16 public class TaskDemo extends Application
17 {
18     private TextArea content = new TextArea("");
19     private Label status = new Label();
20     private ExecutorService executor = Executors.newCachedThreadPool();
21     private Task<Integer> task;
22     private Button open = new Button("Open");
23     private Button cancel = new Button("Cancel");
24
25     public void start(Stage stage)
26     {
27         open.setOnAction(event -> read(stage));
28         cancel.setOnAction(event ->
29             {
30                 if (task != null) task.cancel();
31             });
32         cancel.setDisable(true);
33         stage.setOnCloseRequest(event ->
34             {
35                 if (task != null) task.cancel();
36                 executor.shutdown();
37                 Platform.exit();
38             });
39
40         HBox box = new HBox(10, open, cancel);
41         VBox pane = new VBox(10, content, box, status);
42         pane.setPadding(new Insets(10));
43         stage.setScene(new Scene(pane));
```

(Continues)

Listing 13.21 *(Continued)*

```
44     stage.setTitle("TaskDemo");
45     stage.show();
46 }
47
48 private void read(Stage stage)
49 {
50     if (task != null) return;
51     FileChooser chooser = new FileChooser();
52     chooser.setInitialDirectory(new File("."));
53     File file = chooser.showOpenDialog(stage);
54     if (file == null) return;
55     content.setText("");
56     task = new Task<>()
57     {
58         public Integer call()
59         {
60             int lines = 0;
61             try (Scanner in = new Scanner(file, StandardCharsets.UTF_8))
62             {
63                 while (!isCancelled() && in.hasNextLine())
64                 {
65                     Thread.sleep(10); // simulate work
66                     String line = in.nextLine();
67                     Platform.runLater(() ->
68                         content.appendText(line + "\n"));
69                     lines++;
70                     updateMessage(lines + " lines read");
71                 }
72             }
73             catch (InterruptedException e)
74             {
75                 // task was canceled in sleep
76             }
77             catch (IOException e)
78             {
79                 throw new UncheckedIOException(null, e);
80             }
81             return lines;
82         }
83     };
84     executor.execute(task);
85     task.setOnScheduled(event ->
86     {
87         cancel.setDisable(false);
88         open.setDisable(true);
89     });
```



```

90     task.setOnRunning(event ->
91     {
92         status.setText("Running");
93         status.textProperty().bind(task.messageProperty());
94     });
95     task.setOnFailed(event ->
96     {
97         cancel.setDisable(true);
98         status.textProperty().unbind();
99         status.setText("Failed due to " + task.getException());
100        task = null;
101        open.setDisable(false);
102    });
103    task.setOnCancelled(event ->
104    {
105        cancel.setDisable(true);
106        status.textProperty().unbind();
107        status.setText("Canceled");
108        task = null;
109        open.setDisable(false);
110    });
111    task.setOnSucceeded(event ->
112    {
113        cancel.setDisable(true);
114        status.textProperty().unbind();
115        status.setText("Done reading " + task.getValue() + " lines");
116        task = null;
117        open.setDisable(false);
118    });
119 }
120 }

```

javafx.application.Platform

- static void `runLater(Runnable runnable)`
calls `runnable.run()` on the UI thread.

javafx.concurrent.Task<V>

- protected abstract `V call()`
override this method to carry out the work of the task.
- boolean `cancel()`
cancels this task.

(Continues)

javafx.concurrent.Task<V> (Continued)

- `V getValue()`
yields the value set by the `updateValue` method, after successful completion, the value returned by the call method.
- `Throwable getException()`
yields the exception that terminated the call method, or null if none was thrown.
- `void setOnCancelled(EventHandler<WorkerStateEvent> value)`
- `void setOnFailed(EventHandler<WorkerStateEvent> value)`
- `void setOnRunning(EventHandler<WorkerStateEvent> value)`
- `void setOnScheduled(EventHandler<WorkerStateEvent> value)`
- `void setOnSucceeded(EventHandler<WorkerStateEvent> value)`
sets the event handler for the given worker state event.
- `protected void updateMessage(String message)`
- `protected void updateProgress(double workDone, double max)`
- `protected void updateProgress(long workDone, long max)`
- `protected void updateTitle(String title)`
- `protected void updateValue(V value)`
updates the given property. The `updateProgress` method sets the `workDone` and `totalWork` properties to the arguments and the `progress` property to the ratio of the arguments. Property updates are coalesced and later executed on the the UI thread.

That brings us to the end of this fast-paced introduction to JavaFX. JavaFX has a few rough edges, mostly due to a hurried transformation from the original scripting language. But it is certainly no harder to use than Swing, and it has many more useful and attractive controls than Swing ever had.