CHAPTER 10

GRAPHICAL USER INTERFACES (FX)

© Trout55/iStockphoto.

CHAPTER GOALS

To implement simple graphical user interfaces

To work with buttons, text fields, and other controls

To handle events that are generated by buttons

To write programs that display simple drawings

CHAPTER CONTENTS

In this chapter, you will learn how to write graphical user-interface applications using the JavaFX toolkit. You will be able to produce applications that contain buttons, text components, and drawings—and that react to user input.

© Trout55/iStockphoto.

10.1 Displaying Panes and Controls

A graphical application shows information inside a window with a title bar, as shown in Figure 1. In the following sections, you will learn how to display such a window and how to place user-interface components inside it.

10.1.1 Displaying an Empty Pane

A JavaFX user interface is displayed on a stage.

When Java was first created, it came with a toolkit for graphical user interfaces called the "abstract window toolkit" or AWT. It was abstract in the sense that it that mapped classes such as `Window`, `Button`, and `TextField` to the user-interface elements of Windows, Mac OS, and UNIX. That approach worked fine for simple applications, but complex applications ran into trouble when there were subtle differences in those user interfaces. A few years later, Java provided the "Swing" user-interface toolkit, which solved those problems. Swing took over event handling, and it painted all pixels of the user interface. That approach worked well for many years, but nowadays, user interfaces are no longer painted one pixel at a time, but instead are rendered at tremendous speed by dedicated graphics hardware. The JavaFX toolkit, which is now a part of the Oracle Java distribution and should soon be a part of open-source Java distributions, enables attractive user interfaces on modern hardware.

JavaFX uses a theatrical metaphor to describe an application. The application executes on a *stage*, which represents the window in which the application runs. A stage shows one or more *scenes*. Our simple applications will have only a single scene.

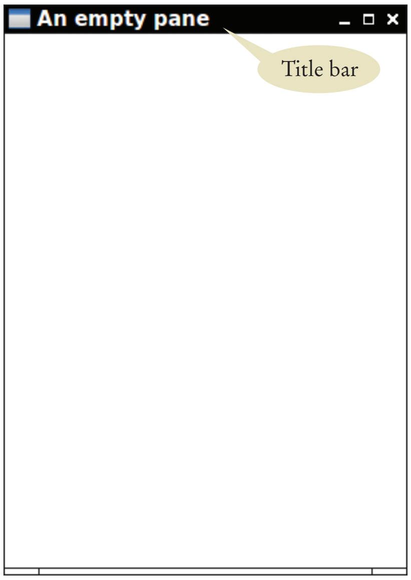


Figure 1
A Window with an Empty Pane

A scene can be populated with any number of items, but they have to be placed in a container, which for us will always be a subclass of the `Pane` class.

Here are the steps that you need to show a pane:

1. Fill in the details of the `start` method. That method is automatically called when a JavaFX program starts. It has a parameter of type `Stage`.

   ```
   public class MyApplication extends Application
   {
      public void start(Stage stage1)
      {
         Populate and show the stage.
      }
   }
   ```

2. Construct an empty pane. We call it the *root pane* because in a real application, it will hold all items that make up the user interface of an application. When a pane is populated with items, it becomes just large enough to hold all of them. If you want a pane to be larger, you need to set its minimum size.

   ```
   Pane root = new Pane();
   final int PANE_WIDTH = 300;
   final int PANE_HEIGHT = 400;
   root.setMinSize(PANE_WIDTH, PANE_HEIGHT);
   ```

3. Make a scene that displays the root pane. Set this scene as the current scene of the stage. If you like, set the title of the window. Then show the stage.

   ```
   Scene scene1 = new Scene(root);
   stage.setScene(scene1);
   stage.setTitle("An empty pane");
   stage.show();
   ```

The simple program below shows all of these steps. It produces the empty pane shown in Figure 1. Note that, unlike all other Java programs, JavaFX programs do not need a `main` method.

sec01_01/EmptyPaneViewer.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class EmptyPaneViewer extends Application
{
   public void start(Stage stage1)
   {
      Pane root = new Pane();
      final int PANE_WIDTH = 300;
      final int PANE_HEIGHT = 400;
      root.setMinSize(PANE_WIDTH, PANE_HEIGHT);

      Scene scene1 = new Scene(root);
      stage1.setScene(scene1);
      stage1.setTitle("An empty pane");
      stage1.show();
   }
}
```

10.1.2 Adding User-Interface Controls to a Pane

When building a graphical user interface, you add controls to the root pane.

An empty pane is not very interesting. You will want to add some user-interface controls, such as buttons and text labels.

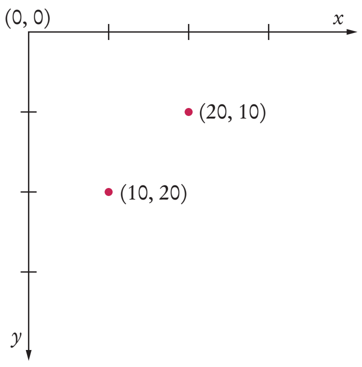
You first construct the controls, providing the text that should appear on them:

```
Button button1 = new Button("Click me!");
Label label1 = new Label("Hello, World!");
```

Then you construct a pane that holds the controls. You can specify any number of controls in the `Pane` constructor. However, if you add controls to a `Pane` object, they get placed on top of each other. For now, we will call the `relocate` method on controls to move them away from each other. In Chapter 11, you will learn a better way of positioning the controls, using layout panes.

```
Pane root = new Pane(button1, label1);
label1.relocate(0, 50);
```

When you call methods such as `relocate`, you need to keep in mind that the coordinate system used by JavaFX is different from the one used in mathematics. The origin (0, 0) is at the upper-left corner of the pane, and the *y*-coordinate grows downward.



Now construct a scene from the root pane, and add it to the stage, as we did in the first example. Figure 2 shows the result. When you run the program, you can click the button, but nothing will happen. You will see in Section 10.2 how to attach an action to a button.

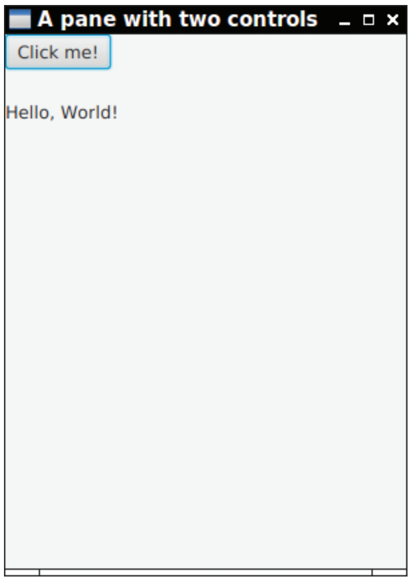


Figure 2 A Scene with Two Controls

sec01_02/FilledPaneViewer.java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class FilledPaneViewer extends Application
{
   public void start(Stage stage1)
   {
      Button button1 = new Button("Click me!");
      Label label1 = new Label("Hello, World!");
      label1.relocate(0, 50);

      Pane root = new Pane(button1, label1);
      final int PANE_WIDTH = 300;
      final int PANE_HEIGHT = 400;
      root.setMinSize(PANE_WIDTH, PANE_HEIGHT);

      Scene scene1 = new Scene(root);
      stage1.setScene(scene1);
      stage1.setTitle("A pane with two controls");
      stage1.show();
   }
}
```

SELF CHECK

1. How do you display a square window with a title that reads “Hello, World!”?
2. How can a program show a pane with two buttons labeled Yes and No?
3. What happens when you omit the call to the `relocate` method in the `FilledPaneViewer` program? Try it out!
4. What happens when you omit the call to the `setMinSize` method in the `FilledPaneViewer` program? Try it out!
5. What happens when you omit the call to the `setScene` method in the `FilledPaneViewer` program? Try it out!

Practice It Now you can try these exercises at the end of the chapter: R10.1, R10.4, E10.1.

10.2 Event Handling

User-interface events include key presses, mouse moves, button clicks, menu selections, and so on.

In an application that interacts with the user through a console window, user input is under control of the program. The program asks the user for input in a specific order. For example, a program might ask the user to supply first a name, then a dollar amount. But the programs that you use every day on your computer don’t work like that. In a program with a modern **graphical user interface**, the *user* is in control. The user can use both the mouse and the keyboard and can manipulate many parts of the user interface in any desired order. For example, the user can enter information into text fields, click buttons, pull down menus, and drag scroll bars in any order. The program must react to the user commands in whatever order they arrive. Having to deal with many possible inputs in random order is quite a bit harder than simply forcing the user to supply input in a fixed order.

Big Java, Late Objects, 2e, Cay Horstmann, © 2017 John Wiley & Sons, Inc. All rights reserved.

In the following sections, you will learn how to write Java programs that can react to user-interface events.

10.2.1 Handling Events

In an event-driven user interface, the program receives an event whenever the user manipulates an input component.

Whenever the user of a graphical program types on the keyboard or uses the mouse anywhere inside one of the windows of a program, the program receives a notification that an event has occurred. For example, whenever the mouse moves a tiny interval over a window, a "mouse move" event is generated. Clicking a button or selecting a menu item generates an "action" event.

© Seriy Tryapitsyn/iStockphoto.

In an event-driven user interface, the program receives an event whenever the user manipulates an input component.

Most programs don't want to be flooded by irrelevant events. For example, when a button is clicked with the mouse, the mouse moves over the button, then the mouse button is pressed, and finally the button is released. Rather than receiving all these mouse events, a program can indicate that it only cares about button clicks, not about the underlying mouse events. On the other hand, if the mouse input is used for drawing shapes on a virtual canvas, a program needs to closely track mouse events.

An event handler describes the actions to be taken when an event occurs.

Every program must indicate which events it needs to receive. It does that by installing **event handler** objects. These objects have one or more methods that will be executed when the events occur.

Event sources report on events. When an event occurs, the event source notifies an event handler.

To install a handler, you need to know the **event source**. The event source is the object that generates a particular event. An event source notifies an event handler when an event occurs by invoking a method on the handler object.

This sounds somewhat abstract, so let's run through an extremely simple program that prints a message whenever a button is clicked. When a button is clicked, it notifies a handler belonging to a class that implements the `EventHandler<ActionEvent>` interface.

The interface `EventHandler<T>` has a single method, `void handle(T event)`. It is your job to supply a class whose `handle` method contains the instructions that you want executed whenever the button is clicked. Here is a very simple example of such a handler class:

sec02_01/ClickHandler.java

```java
import javafx.event.ActionEvent;
import javafx.event.EventHandler;

public class ClickHandler implements EventHandler<ActionEvent>
{
   public void handle(ActionEvent event)
   {
      System.out.println("I was clicked.");
   }
}
```

We ignore the `event` parameter variable of the `handle` method—it contains additional details about the event, such as the time at which it occurred.

Once the handler class has been declared, we need to construct an object of the class and register it with the button as the handler for action events:

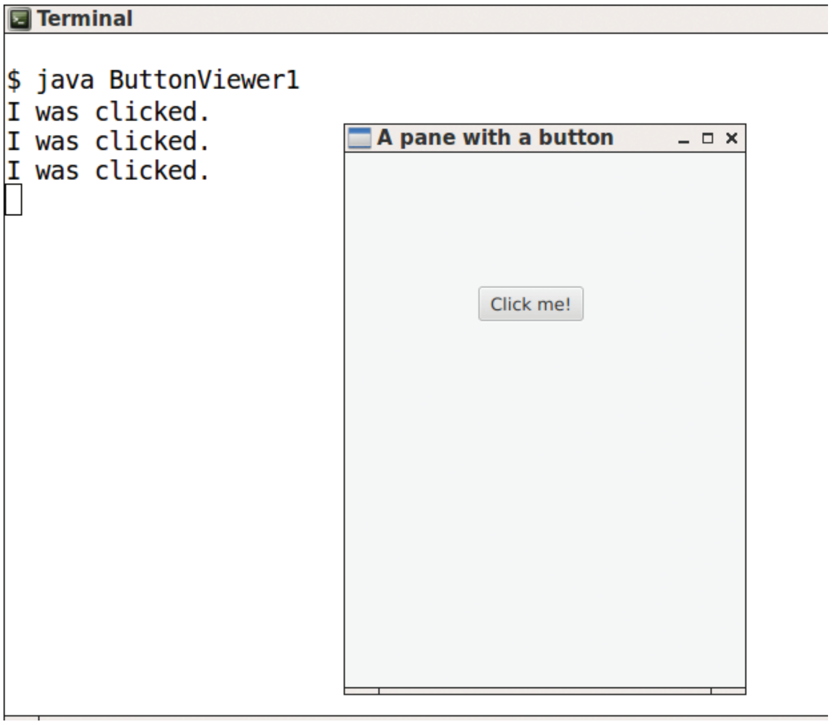
```
EventHandler<ActionEvent> handler = new ClickHandler();
button1.setOnAction(handler);
```

Whenever the button is clicked, the JavaFX event handling mechanism calls

```
handler.handle(event);
```

As a result, the message is printed.

You can test this program out by opening a console window, starting the `ButtonViewer1` program from that console window, clicking the button, and watching the messages in the console window (see Figure 3).



```
$ java ButtonViewer1
I was clicked.
I was clicked.
I was clicked.
```

Figure 3 Implementing a Button Action Event Handler

sec02_01/ButtonViewer1.java

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class ButtonViewer1 extends Application
{
   public void start(Stage stage1)
   {
      Button button1 = new Button("Click me!");
      EventHandler<ActionEvent> handler = new ClickHandler();
      button1.setOnAction(handler);

      Pane root = new Pane(button1);
      final int PANE_WIDTH = 300;
      final int PANE_HEIGHT = 400;
      root.setMinSize(PANE_WIDTH, PANE_HEIGHT);

      Scene scene1 = new Scene(root);
      stage1.setScene(scene1);
      stage1.setTitle("A pane with a button");
      stage1.show();
   }
}
```

10.2.2 Using Lambda Expressions for Handlers

In the preceding section, you saw how to specify button actions. The code for the button action is placed into a handler class. However, it is tedious to define a new class for each handler, particularly when a program has to handle many different events.

Lambda expressions provide a concise syntax for event handlers.

With lambda expressions (see Java 8 Note 9.3), you can express handlers very concisely. You specify only the parameter variable and the body of the handler method. For example, here is how you can provide the button handler from the preceding section:

```
button.setOnAction(
    (ActionEvent event) ->
    {
       System.out.println("I was clicked!");
    });
```

The compiler knows that the `setOnAction` method expects an object of some class implementing the `EventHandler<ActionEvent>` interface. Therefore, the compiler makes that class, constructs an object, and passes it to the `setOnAction` method.

This works for any interface that has a single method. You can even omit the parameter type because the compiler knows it from the interface declaration:

```
button.setOnAction(
    (event) ->
    {
       System.out.println("I was clicked!");
    });
```

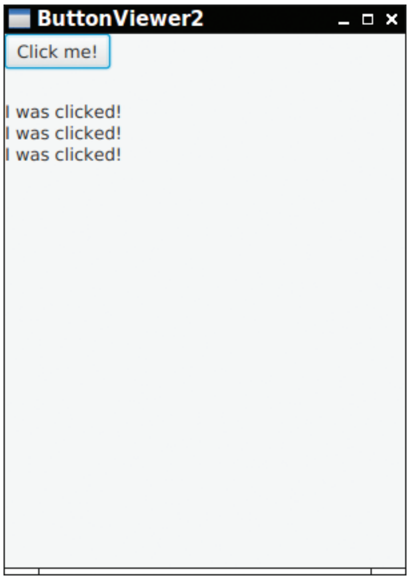
If there is only one parameter variable, you can omit the parentheses around it. If the body has just one method call, you can omit the braces and semicolon. The shortest and most convenient way of denoting this event handler is as follows:

```
button.setOnAction(event -> System.out.println("I was clicked!"));
```

Note that you must specify the parameter variable of the handler method, even if you do not use it in the method body.

Lambda expressions can access variables and methods from the enclosing scope.

There are two advantages to making a handler class into a lambda expression. First, you can put the handler method close to where it is needed, without cluttering up the remainder of the project. Moreover, lambda expressions have a very attractive feature: Their methods can access variables of the enclosing method, and variables and methods of the enclosing class.



```
I was clicked!
I was clicked!
I was clicked!
```

Figure 4 The Button Handler Updates the Label Text

Let's look at an example. Instead of printing the message "I was clicked!", we want to show it in a label, as shown in Figure 4. If we make the handler method into a lambda expression, it can access the `label1` variable and change the text of the label:

```
Label label1 = new Label("");
Button button1 = new Button("Click me!");
button1.setOnAction(event ->
   label1.setText(label1.getText() + "I was clicked!\n"));
```

Having the handler as a regular class would be unattractive—the handler would need to be constructed with a reference to the label (see Exercise ••• E10.7).

There is a technical restriction. When a lambda expression accesses a *local* variable (such as the `label1` variable in this example), then the variable must be *effectively final*. That means, it is either declared as `final`, or it could have been. In this case, the condition is fulfilled. Because `label1` never changes, it could have been declared as `final`. If you want to access a variable from a handler that changes its values, then you need to make the variable into an instance variable of the enclosing class. You will see an example in the following section.

sec02_02/ButtonViewer2.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class ButtonViewer2 extends Application
{
   public void start(Stage stage1)
   {
      Label label1 = new Label("");
      label1.relocate(0, 50);

      Button button1 = new Button("Click me!");
      button1.setOnAction(event ->
            label1.setText(label1.getText() + "I was clicked!\n"));

      Pane root = new Pane(label1, button1);

      final int PANE_WIDTH = 300;
      final int PANE_HEIGHT = 400;
      root.setMinSize(PANE_WIDTH, PANE_HEIGHT);

      Scene scene1 = new Scene(root);
      stage1.setScene(scene1);
      stage1.setTitle("ButtonViewer2");
      stage1.show();
   }
}
```

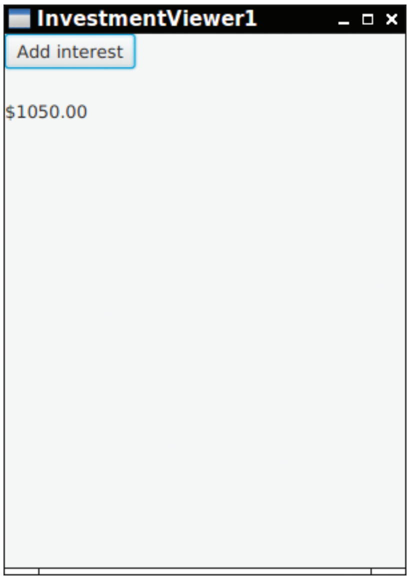
10.2.3 Application: Showing Growth of an Investment

In this section, we will build a practical application with a graphical user interface. A pane displays the amount of money in a bank account. Whenever the user clicks a button, 5 percent interest is added, and the new balance is displayed (see Figure 5).

We need a button and a label for the user interface. We also need to store the current balance.

```
public class InvestmentViewer1 extends Application
{
   private Button addInterestButton;
   private Label resultLabel;
   private double balance;

   private static final double INTEREST_RATE = 5;
   private static final double INITIAL_BALANCE = 1000;
   . . .
}
```



$1050.00

Figure 5
Clicking the Button Grows the Investment

Because the user interface is getting more complex, we set it up in a separate method:

```
private Pane createRootPane()
{
   balance = INITIAL_BALANCE;

   addInterestButton = new Button("Add interest");
   resultLabel = new Label(String.format("$%.2f", balance));
   resultLabel.relocate(0, 50);

   Pane root = new Pane(addInterestButton, resultLabel);
   . . .
   return root;
}
```

Now we are ready for the hard part—the event handler that processes button clicks. Our handler method adds interest and displays the new balance:

```
addInterestButton.setOnAction(event ->
   {
      double interest = balance * INTEREST_RATE / 100;
      balance = balance + interest;
      resultLabel.setText(String.format("$%.2f", balance));
   });
```

Here is the complete program. It demonstrates how to add multiple components to a pane, and how to implement handlers as lambda expressions.

sec02_03/InvestmentViewer1.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class InvestmentViewer1 extends Application
{
   private Button addInterestButton;
   private Label resultLabel;
   private double balance;

   private static final double INTEREST_RATE = 5;
   private static final double INITIAL_BALANCE = 1000;

   public void start(Stage stage1)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      stage1.setScene(scene1);
      stage1.setTitle("InvestmentViewer1");
      stage1.show();
   }

   private Pane createRootPane()
   {
      balance = INITIAL_BALANCE;

      addInterestButton = new Button("Add interest");
      resultLabel = new Label(String.format("$%.2f", balance));
      resultLabel.relocate(0, 50);

      Pane root = new Pane(addInterestButton, resultLabel);
      final int PANE_WIDTH = 300;
      final int PANE_HEIGHT = 400;
      root.setMinSize(PANE_WIDTH, PANE_HEIGHT);

      addInterestButton.setOnAction(event ->
         {
            double interest = balance * INTEREST_RATE / 100;
            balance = balance + interest;
            resultLabel.setText(String.format("$%.2f", balance));
         });
      return root;
   }
}
```

SELF CHECK

6. Which objects are the event source and the event handler in the `ButtonViewer1` program?
7. Why is it legal to assign a `ClickHandler` object to a variable of type `EventHandler<ActionEvent>`?
8. When do you call the `handle` method?
9. What would happen if you declared the `balance` variable in the `ButtonViewer2` program as a local variable of the `createRootPane` method?
10. What would happen if you declared the `resultLabel` variable in the `ButtonViewer2` program as a local variable of the `createRootPane` method?

Practice It Now you can try these exercises at the end of the chapter: R10.7, E10.2, E10.4.

10.3 Processing Text Input

We continue our discussion with graphical user interfaces that accept text input. Most graphical programs collect text input through text components (see Figure 6 and Figure 8). In the following two sections, you will learn how to add text components to a graphical application, and how to read what the user types into them.

10.3.1 Text Fields

Use a `TextField` component for reading a single line of input. Place a `Label` next to each text field.

The `TextField` class provides a text field for reading a single line of text. You can specify an initial text in the constructor.

```
TextField interestRateField = new TextField("" + INITIAL_INTEREST_RATE);
```

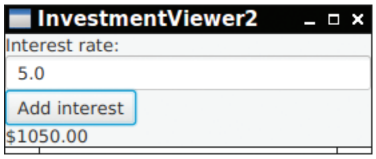


Figure 6 An Application with a Text Field

You can also set the column count—the approximate number of characters that you want to be able to show in the text field.

```
final int COLUMN_COUNT = 20;
interestRateField.setPrefColumnCount(COLUMN_COUNT);
```

Users can type additional characters, but then a part of the contents of the field becomes invisible.

You will want to label each text field so that the user knows what to type into it. Construct a `Label` object for each label:

```
Label rateLabel = new Label("Interest Rate: ");
```

You want to give the user an opportunity to enter all information into the text fields before processing it. Therefore, you should supply a button that the user can press to indicate that the input is ready for processing.

When that button is clicked, its handler should read the user input from each text field, using the `getText` method of the `TextField` class. The `getText` method returns a `String` object. In our sample program, we turn the string into a number, using the `Double.parseDouble` method. After updating the account, we show the balance in another label.

```
addInterestButton.setOnAction(event ->
   {
      double rate = Double.parseDouble(interestRateField.getText());
      double interest = balance * rate / 100;
      balance = balance + interest;
      resultLabel.setText(String.format("$%.2f", balance));
   });
```

For simplicity, this program does not try to handle input errors. It would be better to detect invalid inputs and provide feedback to the user. Exercise •• P10.9 explores this enhancement.

Rather than manually positioning each control, we place them all into a `VBox`, a pane that arranges its contents vertically. You will see how to achieve more sophisticated arrangements in Chapter 11.

```
Pane root = new VBox(rateLabel, interestRateField, addInterestButton, resultLabel);
```

The following application is a useful prototype for a graphical user-interface front end for arbitrary calculations. You can easily modify it for your own needs. Place input controls into the `VBox`. In the `setOnAction` method, carry out the needed calculations. Display the result in a label.

sec03_01/InvestmentViewer2.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.Pane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class InvestmentViewer2 extends Application
{
   private double balance;

   private static final double INITIAL_INTEREST_RATE = 5;
   private static final double INITIAL_BALANCE = 1000;

   public void start(Stage stage1)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      stage1.setScene(scene1);
      stage1.setTitle("InvestmentViewer2");
      stage1.show();
   }

   private Pane createRootPane()
   {
      balance = INITIAL_BALANCE;

      Label rateLabel = new Label("Interest rate:");
      TextField interestRateField = new TextField("" + INITIAL_INTEREST_RATE);
      final int COLUMN_COUNT = 20;
      interestRateField.setPrefColumnCount(COLUMN_COUNT);

      Button addInterestButton = new Button("Add interest");
      Label resultLabel = new Label(String.format("$%.2f", balance));

      Pane pane1 = new VBox(rateLabel, interestRateField,
         addInterestButton, resultLabel);

      addInterestButton.setOnAction(event ->
         {
            double rate = Double.parseDouble(interestRateField.getText());
            double interest = balance * rate / 100;
            balance = balance + interest;
            resultLabel.setText(String.format("$%.2f", balance));
         });
      return pane1;
   }
}
```

10.3.2 Text Areas

Use a TextArea to show multiple lines of text.

In the preceding section, you saw how to construct text fields. A text field holds a single line of text. To display multiple lines of text, use the TextArea class.

After constructing a text area, you can specify the number of desired rows and columns:

```
String initialContents = . . .;
TextArea resultsArea = new TextArea(initialContents);
final int ROW_COUNT = 10;
final int COLUMN_COUNT = 20;
resultsArea.setPrefRowCount(ROW_COUNT);
resultsArea.setPrefColumnCount(COLUMN_COUNT);
```

© Kyoungil Jeon/iStockphoto.

You can use a text area for reading or displaying multi-line text.

Use the setText method to set the text of a text field or text area. The appendText method adds text to the end of a text area. Use newline characters to separate lines, like this:

```
resultsArea.appendText(String.format("$%.2f%n", balance));
```

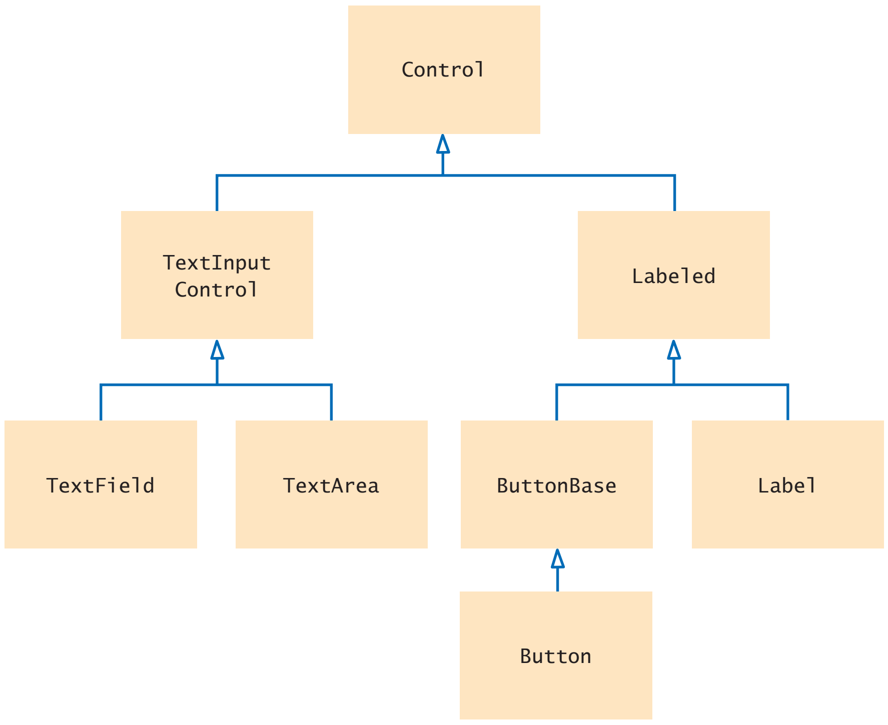


Figure 7 A Part of the Hierarchy of JavaFX User-Interface Controls

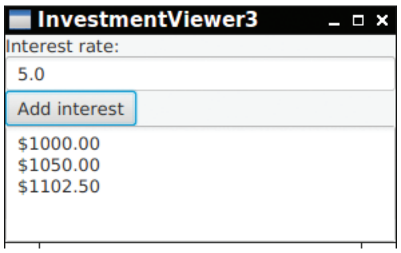
If you want to use a text field or text area for display purposes only, call the `setEditable` method like this:

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

Now the user can no longer edit the contents of the field, but your program can still call `setText` and `appendText` to change it. Why not just use a label instead of a text area for displaying results? A user can copy the contents of a text area, but not that of a label.

As shown in Figure 7, the `TextField` and `TextArea` classes are subclasses of the class `TextInputControl`. The methods `setText`, `appendText`, and `setEditable` are declared in the `TextInputControl` class and inherited by `TextField` and `TextArea`.

The following sample program puts these concepts together. A user can enter numbers into the interest rate text field and then click the "Add Interest" button. The interest rate is applied, and the updated balance is appended to the text area. The text area is not editable.



```
$1000.00
$1050.00
$1102.50
```

Figure 8 The Investment Application with a Text Area

This program is similar to the previous investment viewer program, but it keeps track of all the bank balances, not just the last one.

sec03_02/InvestmentViewer3

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.Pane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class InvestmentViewer3 extends Application
{
   private double balance;

   private static final double INITIAL_INTEREST_RATE = 5;
   private static final double INITIAL_BALANCE = 1000;

   public void start(Stage stage1)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      stage1.setScene(scene1);
      stage1.setTitle("InvestmentViewer3");
      stage1.show();
   }

   private Pane createRootPane()
   {
      balance = INITIAL_BALANCE;

      Label rateLabel = new Label("Interest rate:");
      TextField interestRateField = new TextField("" + INITIAL_INTEREST_RATE);

      Button addInterestButton = new Button("Add interest");
      String initialContents = String.format("$%.2f%n", balance);
      TextArea resultsArea = new TextArea(initialContents);
      final int ROW_COUNT = 10;
      final int COLUMN_COUNT = 20;
      resultsArea.setPrefRowCount(ROW_COUNT);
      resultsArea.setPrefColumnCount(COLUMN_COUNT);
      resultsArea.setEditable(false);

      Pane panel1 = new VBox(rateLabel, interestRateField,
         addInterestButton, resultsArea);

      addInterestButton.setOnAction(event ->
      {
         double rate = Double.parseDouble(interestRateField.getText());
         double interest = balance * rate / 100;
         balance = balance + interest;
         resultsArea.appendText(String.format("$%.2f%n", balance));
      });
      return panel1;
   }
}
```

11. What happens if you omit the first `Label` object in the `InvestmentViewer2` program of Section 10.3.1?
12. If a text field holds an integer, what expression do you use to read its contents?
13. What is the difference between a text field and a text area?
14. Why did the `InvestmentViewer3` program call `resultsArea.setEditable(false)`?
15. Could you have used a `Label` instead of a `TextArea` in the `InvestmentViewer3` program?

Practice It Now you can try these exercises at the end of the chapter: R10.13, E10.9, E10.10.

10.4 Creating Drawings

You often want to include simple drawings such as diagrams or charts in your programs. The following sections show how you can produce drawings with JavaFX.

© Alexey Avdeev/iStockphoto.

You can make simple drawings out of lines, rectangles, and circles.

10.4.1 Drawing on a Pane

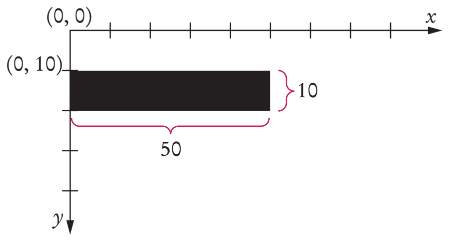
You can add rectangles and other nodes to a pane.

We start out with a simple bar chart (see Figure 9) that is composed of three rectangles.

You construct a rectangle by giving its top-left corner, followed by its width and height:

```
Rectangle rect1 = new Rectangle(0, 10, 50, 10);
   // A rectangle with top-left corner (0, 10), width 50, and height 10
```

Recall that in the coordinate system used by JavaFX, the origin is at the top-left corner of the frame and the y-axis grows downward.



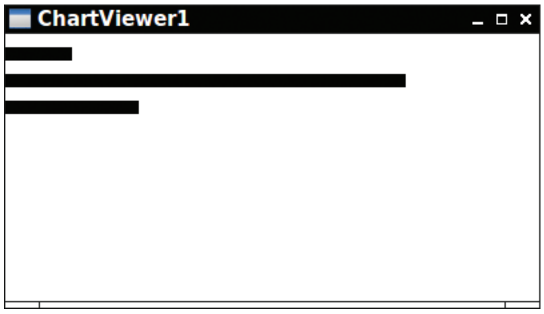


Figure 9
Drawing a Bar Chart

To display the rectangle, you can add it to a pane. Call the `getChildren` method, which yields a list of *nodes*—the common superclass for all objects that can be added to a pane. The `Rectangle` class extends the `Node` class, as does the `Control` class that is the superclass for all JavaFX controls. Given the list of children, you add one at a time:

```
root.getChildren().add(rect1);
root.getChildren().add(rect2);
root.getChildren().add(rect3);
```

Or you can add them all together:

```
root.getChildren().addAll(rect1, rect2, rect3);
```

You add the pane to a scene in the usual way. Here is the complete program.

sec04_01/ChartViewer1.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class ChartViewer1 extends Application
{
   public void start(Stage stage1)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      stage1.setScene(scene1);
      stage1.setTitle("ChartViewer1");
      stage1.show();
   }

   private Pane createRootPane()
   {
      Rectangle rect1 = new Rectangle(0, 10, 50, 10);
      Rectangle rect2 = new Rectangle(0, 30, 300, 10);
      Rectangle rect3 = new Rectangle(0, 50, 100, 10);

      Pane root = new Pane();
      root.getChildren().addAll(rect1, rect2, rect3);
      final int PANE_WIDTH = 400;
      final int PANE_HEIGHT = 200;
      root.setMinSize(PANE_WIDTH, PANE_HEIGHT);

      return root;
   }
}
```

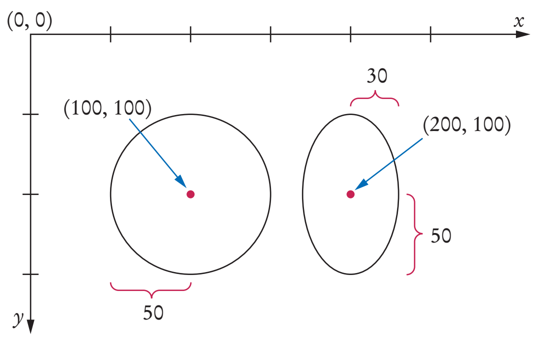
10.4.2 Shapes, Text, Images, and Colors

Circles, ellipses, lines, text, and image views are nodes.

In the preceding section, you learned how to write a program that draws rectangles. Now we turn to additional graphical elements that allow you to draw quite a few interesting pictures.

To construct a circle or ellipse, you specify the center and radius, or, for an ellipse, the major and minor radius.

```
Circle shape1 = new Circle(100, 100, 50);
Ellipse shape2 = new Ellipse(200, 100, 30, 50);
```



To construct a line, you provide the *x*- and *y*-coordinates of the start and end points:

```
Line shape3 = new Line(0, 0, 50, 100);
```

For drawing text, you can simply use a `Label`, and use the `relocate` method to position it:

```
Label message = new Label("Message");
message.relocate(xleft, ytop);
```

You can set the font of the label:

```
Font labelFont = new Font("Serif", 36);
label.setFont(labelFont);
```

For finer control of text placement, or for colored text, use the `Text` class instead—see Special Topic 10.1.

The `ImageView` class lets you add images to your drawing. Construct an image view object from the image file location, and use the `relocate` method to position the image. Exercise •• E10.17 explores options for specifying the location.

```
ImageView cat = new ImageView("cat.jpeg");
cat.relocate(xleft, ytop);
```

All of these classes extend the `Node` class—see Figure 10. Remember to add the `Node` objects to the children of the pane.

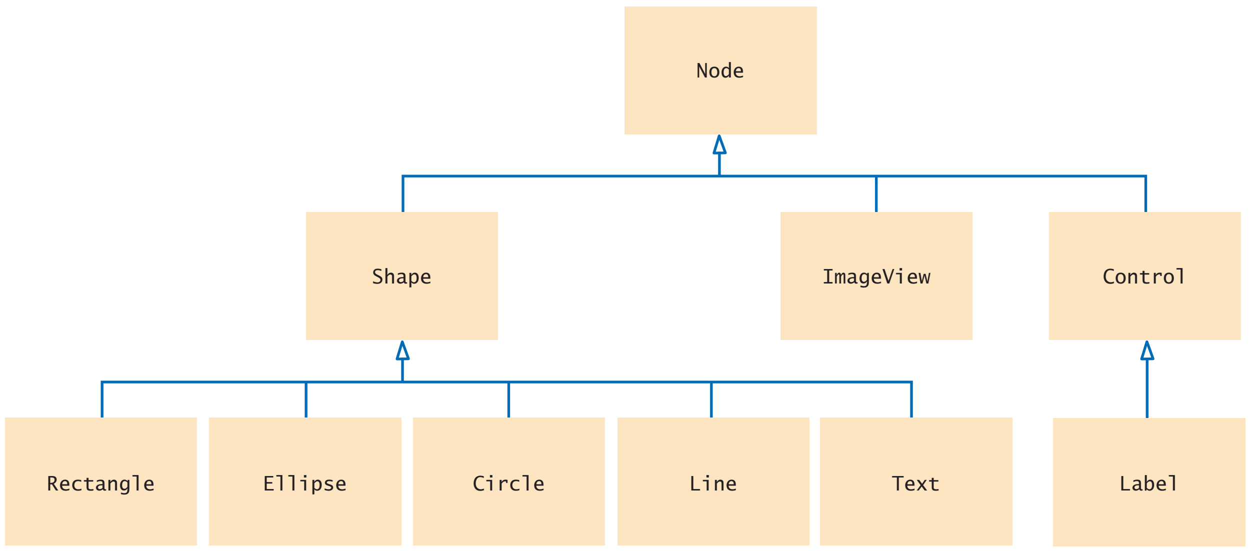


Figure 10 Node Classes in JavaFX

By default, lines and text are drawn with a black *stroke*. Rectangles, circles, and ellipses have a black *fill*. You can use other colors, and you can also stroke the outlines of filled shapes.

```
Rectangle rect = new Rectangle(5, 5, 20, 20);
rect.setStroke(Color.BLACK);
rect.setFill(Color.RED);
```

To draw a rectangle, circle, or ellipse with a hollow interior, set the fill to `Color.TRANSPARENT`:

```
Rectangle hollowRect = new Rectangle(5, 5, 20, 20);
hollowRect.setStroke(Color.BLACK);
hollowRect.setFill(Color.TRANSPARENT);
```

You can specify a color by name, as in the preceding examples. Table 1 shows the most common names. You can also use any of the standard HTML color names, such as `Color.LIGHTSALMON`. Alternatively, you can specify a color by the amounts of the primary colors—red, green, and blue—that make up the color. The amounts are given as integers between `0` (primary color not present) and 255 (maximum amount present). For example,

```
Color magenta = Color.rgb(255, 0, 255);
```

constructs a `Color` object with maximum red, no green, and maximum blue, yielding a purple color called magenta.

Table 1 Commonly Used Colors

Color		RGB Values
`Color.BLACK`	⬛	0, 0, 0
`Color.BLUE`	🟦	0, 0, 255
`Color.CYAN`	🟦	0, 255, 255
`Color.GRAY`	◼	128, 128, 128
`Color.DARKGRAY`	◼	64, 64, 64
`Color.LIGHTGRAY`	◻	192, 192, 192
`Color.GREEN`	🟩	0, 255, 0
`Color.MAGENTA`	🟪	255, 0, 255
`Color.ORANGE`	🟧	255, 200, 0
`Color.PINK`	🟥	255, 175, 175
`Color.RED`	🟥	255, 0, 0
`Color.WHITE`	⬜	255, 255, 255
`Color.YELLOW`	🟨	255, 255, 0

The following program puts all these shapes to work, creating a simple chart (see Figure 11).

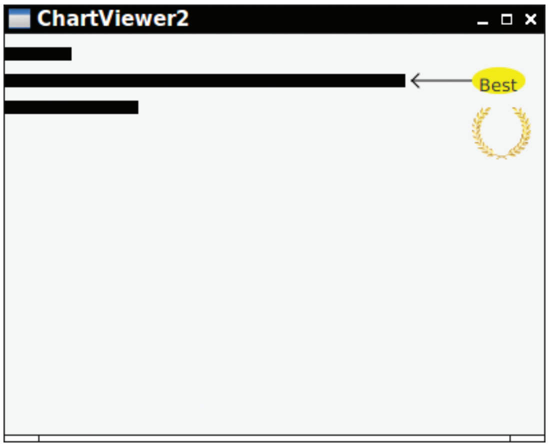


Figure 11 A Chart Decorated with Shapes, Text, and an Image

sec04_02/ChartViewer2.java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.image.ImageView;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Ellipse;
import javafx.scene.shape.Line;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class ChartViewer2 extends Application
{
   public void start(Stage stage1)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      stage1.setScene(scene1);
      stage1.setTitle("ChartViewer2");
      stage1.show();
   }

   private Pane createRootPane()
   {
      Pane root = new Pane();
      final int PANE_WIDTH = 400;
      final int PANE_HEIGHT = 300;
      root.setMinSize(PANE_WIDTH, PANE_HEIGHT);

      // Add bars

      root.getChildren().add(new Rectangle(0, 10, 50, 10));
      root.getChildren().add(new Rectangle(0, 30, 300, 10));
      root.getChildren().add(new Rectangle(0, 50, 100, 10));

      // Add arrow

      root.getChildren().add(new Line(350, 35, 305, 35));
      root.getChildren().add(new Line(305, 35, 310, 30));
      root.getChildren().add(new Line(305, 35, 310, 40));

      Ellipse highlight = new Ellipse(370, 35, 20, 10);
      highlight.setFill(Color.YELLOW);
      Label text = new Label("Best");
      text.relocate(355, 30);
      ImageView laurel = new ImageView("laurel.png");
      laurel.relocate(340, 50);

      root.getChildren().addAll(highlight, text, laurel);

      return root;
   }
}
```

10.4.3 Application: Visualizing the Growth of an Investment

In this section, we will add a bar chart to the investment program of Section 10.3. Whenever the user clicks on the "Add Interest" button, another bar is added to the bar chart (see Figure 12).

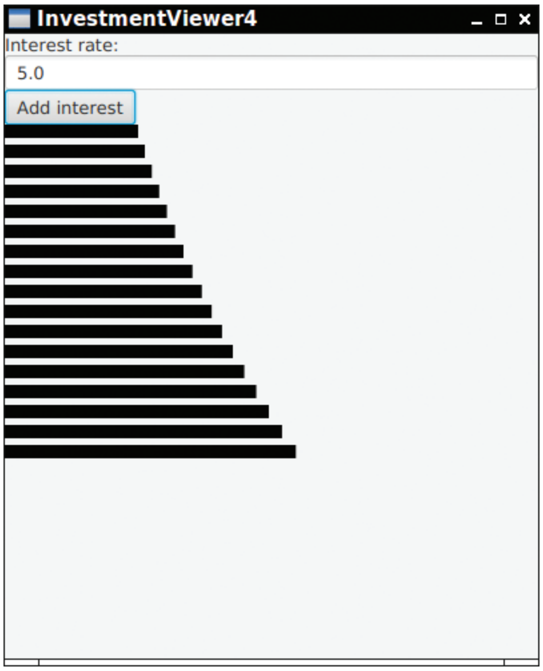


Figure 12 Clicking on the "Add Interest" Button Adds a Bar to the Chart

The chart program of the preceding sections produced a fixed bar chart. We will develop an improved version that can draw a chart with any values.

Make a new class for a drawing that can be reused.

Whenever you produce a complex drawing that is meaningful on its own, it is a good idea to place it in its own class. For our application, we make a bar chart:

```
public class BarChart1 extends Pane
{
   . . .
   public void addValue(double value)
   {
      . . .
   }
}
```

The bar chart extends `Pane` because the `Pane` class knows how to manage a collection of child nodes. When adding a new value, we add a rectangle to the children:

```
public void addValue(double value)
{
   final int GAP = 5;
   final int BAR_HEIGHT = 10;
   int bars = getChildren().size();
   getChildren().add(new Rectangle(0, (BAR_HEIGHT + GAP) * bars,
         value, BAR_HEIGHT));
}
```

We make a separate class for the application:

```
public class InvestmentViewer4 extends Application
{
   . . .
   private Pane createRootPane()
   {
      . . .
   }
}
```

In the `createRootPane` method, we create a label and text field for the interest rate, a button to add the interest, and an instance of `BarChart1`:

```
private Pane createRootPane()
{
   Label rateLabel = new Label("Interest rate:");
   TextField interestRateField = new TextField();
   Button addInterestButton = new Button("Add interest");
   BarChart1 chart = new BarChart1();
   Pane root = new VBox(rateLabel, interestRateField, addInterestButton, chart);
   . . .
   return root;
}
```

As you can see, we treat the bar chart just like any other element that can be added to a pane.

In the button handler, we compute the new balance and add another bar to the chart:

```
addInterestButton.setOnAction(event ->
   {
      double rate = Double.parseDouble(interestRateField.getText());
      double interest = balance * rate / 100;
      balance = balance + interest;
      chart.addValue(balance);
   });
```

That’s all that is required to add a diagram to an application. Here is the code for the chart and viewer classes.

sec04_03/BarChart1.java

```java
import javafx.scene.layout.Pane;
import javafx.scene.shape.Rectangle;

public class BarChart1 extends Pane
{
   public BarChart1()
   {
      final int PANE_WIDTH = 400;
      final int PANE_HEIGHT = 400;
      setMinSize(PANE_WIDTH, PANE_HEIGHT);
   }

   public void addValue(double value)
   {
      final int GAP = 5;
      final int BAR_HEIGHT = 10;
      int bars = getChildren().size();
      getChildren().add(new Rectangle(0, (BAR_HEIGHT + GAP) * bars,
            value, BAR_HEIGHT));
   }
}
```

sec04_03/InvestmentViewer4.java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.Pane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class InvestmentViewer4 extends Application
{
   private double balance;

   private static final double INITIAL_INTEREST_RATE = 5;
   private static final double INITIAL_BALANCE = 100;

   public void start(Stage stage1)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      stage1.setScene(scene1);
      stage1.setTitle("InvestmentViewer4");
      stage1.show();
   }

   private Pane createRootPane()
   {
      balance = INITIAL_BALANCE;

      Label rateLabel = new Label("Interest rate:");
      TextField interestRateField = new TextField();
```

Big Java, Late Objects, 2e, Cay Horstmann, © 2017 John Wiley & Sons, Inc. All rights reserved.

```

      Button addInterestButton = new Button("Add interest");
      BarChart1 chart = new BarChart1();

      Pane root = new VBox(rateLabel, interestRateField,
         addInterestButton, chart);

      interestRateField.setText("" + INITIAL_INTEREST_RATE);
      chart.addValue(INITIAL_BALANCE);
      addInterestButton.setOnAction(event ->
         {
            double rate = Double.parseDouble(interestRateField.getText());
            double interest = balance * rate / 100;
            balance = balance + interest;
            chart.addValue(balance);
         });
      return root;
   }
}
```

SELF CHECK

16. How do you modify the program in Section 10.4.1 to draw two squares?
17. Give instructions to draw a red circle with center (100, 100) and radius 25.
18. Give instructions to draw a letter “V” by drawing two line segments.
19. Give instructions to draw a string consisting of the letter “V”.
20. What are the RGB color values of `Color.BLUE`?

Practice It Now you can try these exercises at the end of the chapter: R10.21, E10.15, E10.16.

Special Topic 10.1

The Text Class

If you need more control over text than the `Label` class gives you, use the `Text` class instead.
You can change the color of a `Text` node by setting the stroke color:

```
Text caution = new Text("Caution");
caution.setStroke(Color.RED);
```

When you specify the text position, the text is placed so that its basepoint is at the given location (see Figure 13).

```
Text message = new Text(xleft, ybase, "Message");
```

This is useful when you want to line up text in different fonts. In contrast, when you call `relocate(x, y)` on a `Text` object, the coordinates specify the top-left corner, as they do for a `Label` object.

To find the extent of a `Text` node, call the `getBoundsInParent` method. You can retrieve the top, the width, and the total height from the `Bounds` object.

```
Bounds b = message.getBoundsInParent();
double ytop = b.getMinY();
double messageWidth = b.getWidth();
double messageHeight = b.getHeight();
```

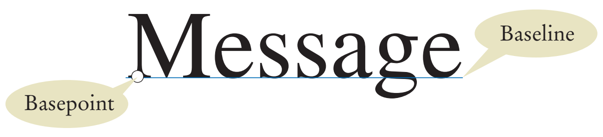
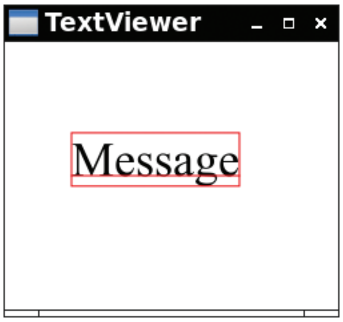


Figure 13
Basepoint and Baseline

The following sample program uses these values to illustrate the extent and the baseline of a Text node.



special_topic_1/TextViewer.java

```java
import javafx.application.Application;
import javafx.geometry.Bounds;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Line;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class TextViewer extends Application
{
   public void start(Stage stage1)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      stage1.setScene(scene1);
      stage1.setTitle("TextViewer");
      stage1.show();
   }

   private Pane createRootPane()
   {
      double xleft = 50;
      double ybase = 100;
      Text message = new Text(xleft, ybase, "Message");
      message.setFont(new Font("Times New Roman", 36));

      Bounds b = message.getBoundsInParent();
      double ytop = b.getMinY();
      double width = b.getWidth();
      Rectangle rect = new Rectangle(xleft, ytop, width, b.getHeight());
      rect.setFill(Color.TRANSPARENT);
      rect.setStroke(Color.RED);
      Line baseLine = new Line(xleft, ybase, xleft + width, ybase);
      baseLine.setStroke(Color.RED);

      Pane root = new Pane(message, rect, baseLine);

      final int PANE_WIDTH = 250;
      final int PANE_HEIGHT = 200;
      root.setMinSize(PANE_WIDTH, PANE_HEIGHT);
```

```

         return root;
      }
   }
```

HOW TO 10.1 Drawing Graphical Shapes

Suppose you want to write a program that displays graphical shapes such as cars, aliens, charts, or any other images that can be obtained from rectangles, lines, and ellipses. These instructions give you a step-by-step procedure for decomposing a drawing into parts and implementing a program that produces the drawing.

Problem Statement Create a program to draw a national flag.

Step 1 Determine the shapes that you need for the drawing.

You can use the following shapes:

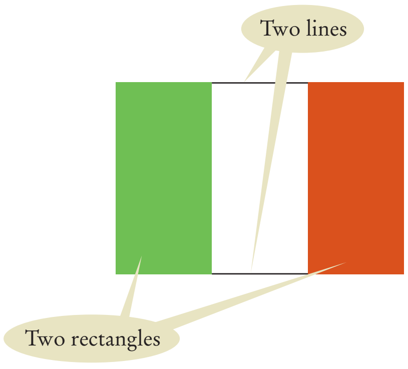
- Squares and rectangles
- Circles and ellipses
- Lines
- Text and images

The outlines of these shapes can be drawn in any color, and you can fill the insides of these shapes with any color. You can use text to label parts of your drawing. You can add images from image files, but that is only helpful if the image can be used "as is" without any changes.

Some national flag designs consist of three equally wide sections of different colors, side by side, as in the Italian flag shown below.

Punchstock.

You could draw such a flag using three rectangles. But if the middle rectangle is white, as it is in the flag of Italy (green, white, red), it is easier and looks better to draw a line on the top and bottom of the middle portion:



Step 2 Find the coordinates for the shapes.

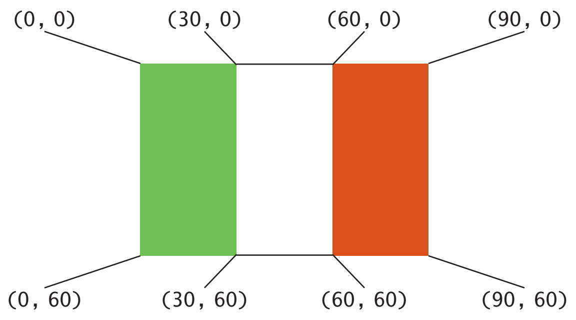
You now need to find the exact positions for the geometric shapes.

- For rectangles, you need the *x*- and *y*-position of the top-left corner, the width, and the height.
- For circles, you need the center and radius; for ellipses, the center and major and minor radii.
- For lines, you need the *x*- and *y*-positions of the starting point and the end point.
- For text labels and images, you need the top-left corner.

Many flags, such as the flag of Italy, have a width : height ratio of 3 : 2. (You can often find exact proportions for a particular flag by doing a bit of Internet research on one of several Flags of the World sites.) For example, if you make the flag 90 pixels wide, then it should be 60 pixels tall. (Why not make it 100 pixels wide? Then the height would be 100 · 2 / 3 ≈ 67, which seems more awkward.)

Because we can relocate a pane anywhere we want by calling the `relocate` method, we can assume for now that the top-left corner is (0, 0).

Now you can compute the coordinates of all the important points of the shape:



Step 3 Write Java statements to draw the shapes.

In our example, there are two rectangles and two lines:

```
Rectangle left = new Rectangle(0, 0, 30, 60);
left.setFill(Color.GREEN);
Rectangle right = new Rectangle(60, 0, 30, 60);
right.setFill(Color.RED);
Line top = new Line(30, 0, 60, 0);
Line bottom = new Line(30, 60, 60, 60);
```

If you are more ambitious, then you can express the coordinates in terms of a few variables. In the case of the flag, we have arbitrarily chosen the top-left corner and the width. All other coordinates follow from those choices. If you decide to follow the ambitious approach, then the rectangles and lines are determined as follows:

```
double height = width * 2 / 3;
Rectangle left = new Rectangle(0, 0, width / 3, height);
left.setFill(Color.GREEN);
Rectangle right = new Rectangle(width * 2 / 3, 0, width / 3, height);
right.setFill(Color.RED);
Line top = new Line(width / 3, 0, width * 2 / 3, 0);
Line bottom = new Line(width / 3, height, width * 2 / 3, height);
```

Step 4 Consider using methods or classes for repetitive steps.

Do you need to draw more than one flag? Perhaps with different sizes? Then it is a good idea to design a method or class, so you won't have to repeat the same drawing instructions.

For example, you can provide a class

```
public class ItalianFlag extends Pane
{
   public ItalianFlag(double width)
   {
      . . .
      getChildren().addAll(left, right, top, bottom);
   }
}
```

Place the instructions from the preceding step into the constructor. Then you can call

```
ItalianFlag flag1 = new ItalianFlag(150);
ItalianFlag flag2 = new ItalianFlag(300);
flag2.relocate(100, 200);
root.getChildren().addAll(flag1, flag2);
```

in the startup code of your JavaFX application class.

Step 5 Write the application class.

Provide a viewer class that extends `Application`, with a `start` method in which you construct a stage, scene, and root pane. As always, it is a good idea to use a helper method for setting up the root pane.

```
public class ItalianFlagViewer extends Application
{
   public void start(Stage stage1)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      stage1.setScene(scene1);
      stage1.setTitle("ItalianFlagViewer");
      stage1.show();
   }

   private Pane createRootPane()
   {
      Pane root = new Pane();
      Add shapes as children.
      return root;
   }
}
```

how_to_1/ItalianFlag.java

```
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Line;
import javafx.scene.shape.Rectangle;

public class ItalianFlag extends Pane
{
   public ItalianFlag(double width)
   {
      double height = width * 2 / 3;
      Rectangle left = new Rectangle(0, 0, width / 3, height);
      left.setFill(Color.GREEN);
      Rectangle right = new Rectangle(width * 2 / 3, 0, width / 3, height);
      right.setFill(Color.RED);
      Line top = new Line(width / 3, 0, width * 2 / 3, 0);
      Line bottom = new Line(width / 3, height, width * 2 / 3, height);

      getChildren().addAll(left, right, top, bottom);
   }
}
```

how_to_1/ItalianFlagViewer.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class ItalianFlagViewer extends Application
{
   public void start(Stage stage1)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      stage1.setScene(scene1);
      stage1.setTitle("ItalianFlagViewer");
      stage1.show();
   }

   private Pane createRootPane()
   {
      Pane root = new Pane();
      final int PANE_WIDTH = 400;
      final int PANE_HEIGHT = 300;
      root.setMinSize(PANE_WIDTH, PANE_HEIGHT);

      ItalianFlag flag1 = new ItalianFlag(150);
      ItalianFlag flag2 = new ItalianFlag(300);
      flag2.relocate(100, 200);
      root.getChildren().addAll(flag1, flag2);
      return root;
   }
}
```

WORKED EXAMPLE 10.1 A Bar Chart Creator

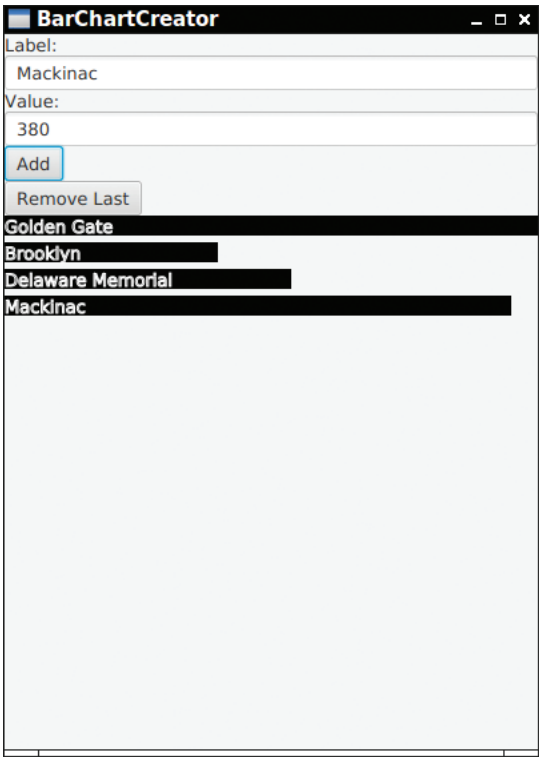
In this Worked Example, we will develop a simple program for creating bar charts. The user enters labels and values for the bars, and the program displays the chart. We also allow the user to fix mistakes by removing the last bar. Admittedly, the user interface is a bit limited. Worked Example 11.1 will improve on it, allowing users to use the mouse to edit the chart.

Our program needs the following user interface elements:

- A text field for entering a bar label
- A text field for entering a bar value
- A button for adding a new bar with the given label and value
- A button for removing the last bar
- A pane for drawing the chart

We will construct the controls and the chart as follows in the `createRootPane` method:

```
public class BarChartCreator extends Application
{
    . . .
   private Pane createRootPane()
   {
      TextField labelField = new TextField("");
      TextField valueField = new TextField("");
      Button addButton = new Button("Add");
      Button removeLastButton = new Button("Remove Last");
      BarChart chart = new BarChart();
      . . .
   }
}
```



The bar chart pane class needs to provide two methods that support the button commands:

```
public void append(String label, double value)
public void removeLast()
```

For now, let us assume that those methods have been implemented. We need to provide handlers that call them:

```
addButton.setOnAction(event ->
   {
      chart.append(labelField.getText(), Double.parseDouble(valueField.getText()));
   });

removeLastButton.setOnAction(event ->
   {
      chart.removeLast();
   });
```

The remainder of the viewer class is straightforward. In the `createRootPane` method, we return a `VBox` with all nodes:

```
return new VBox(
   new Label("Label:"),
   labelField,
   new Label("Value:"),
   valueField,
   addButton,
   removeLastButton,
   chart);
```

Then we set up the stage and scene with the same code that was used for all examples in this chapter.

Now let us turn to the bar chart itself. Unlike the chart of Section 10.4.3, this chart draws the labels in addition to the bars. A bar chart consists of multiple bars, each of which has a label and a value. It is best to make a class for a bar that holds both the label and the value.

```
public class Bar extends Pane
{
   public static final int HEIGHT = 15;

   public Bar(String label, double value)
   {
      Rectangle barRectangle = new Rectangle(0, 0, value, HEIGHT);

      Text barText = new Text(label);
      barText.relocate(0, 0);
      barText.setStroke(Color.WHITE);

      getChildren().addAll(barRectangle, barText);
   }
}
```

Here, we use the `Text` class instead of the `Label` class so that we can change the text color.

Then the chart pane has as its children a list of bars. The `append` and `removeLast` methods add and remove a bar.

```
public class BarChart extends Pane
{
   . . .
   public void append(String label, double value)
   {
      final int GAP = 5;
      int bars = getChildren().size();
      Bar bar1 = new Bar(label, value);
      bar1.relocate(0, (Bar.HEIGHT + GAP) * bars);
      getChildren().add(bar1);
   }

   public void removeLast()
   {
      int bars = getChildren().size();
      if (bars > 0)
      {
         getChildren().remove(bars - 1);
      }
   }
}
```

In the constructor, we set a minimum size, so that the chart occupies some space even if there are no bars inside:

```
public BarChart()
{
   final int PANE_WIDTH = 400;
   final int PANE_HEIGHT = 400;
   setMinSize(PANE_WIDTH, PANE_HEIGHT);
}
```

It is worth reflecting on the division of labor between the classes of this application. The `BarChart` class knows how to draw a chart, and it has methods for modifying the chart data. But it has no notion of a user interface. The same class could be used if we had a different user interface, perhaps with voice recognition instead of the text fields and buttons.

The `BarChartCreator` class, on the other hand, is all about the user interface. It handles the text fields and buttons. As soon as it knows what the user wants to do, it hands the work off to the `BarChart` class.

This is a useful division of labor, giving you guidance if the program needs to be enhanced. For a fancier rendering of the chart, improve the chart pane. For more control over the chart's appearance, add user-interface controls to the application class.

worked_example_1/BarChartCreator.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.Pane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class BarChartCreator extends Application
{
   public void start(Stage stage1)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      stage1.setScene(scene1);
      stage1.setTitle("BarChartCreator");
      stage1.show();
   }

   private Pane createRootPane()
   {
      TextField labelField = new TextField("");
      TextField valueField = new TextField("");
      Button addButton = new Button("Add");
      Button removeLastButton = new Button("Remove Last");
      BarChart chart = new BarChart();
      addButton.setOnAction(event ->
         {
            chart.append(labelField.getText(),
               Double.parseDouble(valueField.getText()));
         });
      removeLastButton.setOnAction(event ->
         {
            chart.removeLast();
         });

      return new VBox(
         new Label("Label:"),
         labelField,
         new Label("Value:"),
         valueField,
         addButton,
         removeLastButton,
         chart);
   }
}
```

worked_example_1/BarChart.java

```
import javafx.scene.layout.Pane;

public class BarChart extends Pane
{
   public BarChart()
   {
      final int PANE_WIDTH = 400;
      final int PANE_HEIGHT = 400;
      setMinSize(PANE_WIDTH, PANE_HEIGHT);
   }

   public void append(String label, double value)
   {
      final int GAP = 5;
      int bars = getChildren().size();
      Bar bar1 = new Bar(label, value);
      bar1.relocate(0, (Bar.HEIGHT + GAP) * bars);
      getChildren().add(bar1);
   }

   public void removeLast()
   {
      int bars = getChildren().size();
      if (bars > 0)
      {
         getChildren().remove(bars - 1);
      }
   }
}
```

worked_example_1/Bar.java

```
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Text;

public class Bar extends Pane
{
   public static final int HEIGHT = 15;

   public Bar(String label, double value)
   {
      Rectangle barRectangle = new Rectangle(0, 0, value, HEIGHT);

      Text barText = new Text(label);
      barText.relocate(0, 0);
```

```
        barText.setStroke(Color.WHITE);

        getChildren().addAll(barRectangle, barText);
    }
}
```

Chapter Summary

Display panes and add controls to panes.

- A JavaFX user interface is displayed on a stage.
- When building a graphical user interface, you add controls to the root pane.

Explain the event concept and handle button events.

- User-interface events include key presses, mouse moves, button clicks, menu selections, and so on.
- In an event-driven user interface, the program receives an event whenever the user manipulates an input component.
- An event handler describes the actions to be taken when an event occurs.
- Event sources report on events. When an event occurs, the event source notifies an event handler.
- Lambda expressions provide a concise syntax for event handlers.
- Lambda expressions can access variables and methods from the enclosing scope.

Use text controls for reading text input.

- Use a `TextField` component for reading a single line of input. Place a `Label` next to each text field.
- Use a `TextArea` to show multiple lines of text.

Create simple drawings with rectangles, circles, lines, and text.

- You can add rectangles and other nodes to a pane.
- Circles, ellipses, lines, text, and image views are nodes.
- Make a new class for a drawing that can be reused.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

```
javafx.application.Application
   start
javafx.collections.ObservableList<E>
   addAll
javafx.event.EventHandler<T>
   handle
javafx.geometry.Bounds
   getHeight
   getMinX
   getMinY
   getWidth
javafx.scene.Parent
   getChildren
javafx.scene.Node
   relocate
javafx.scene.Scene
javafx.scene.control.Button
javafx.scene.control.ButtonBase
   setOnAction
javafx.scene.control.Label
javafx.scene.control.Labeled
   getText
   setFont
   setText
javafx.scene.control.TextArea
   setPrefColumnCount
   setPrefRowCount
javafx.scene.control.TextField
   setPrefColumnCount
javafx.scene.control.TextInputControl
   appendText
   getText
   setText
   setEditable
javafx.scene.image.ImageView
javafx.scene.layout.Pane
javafx.scene.layout.Region
   setMinSize
javafx.scene.layout.VBox
javafx.scene.paint.Color
   rgb
javafx.scene.shape.Rectangle
javafx.scene.shape.Ellipse
javafx.scene.shape.Circle
javafx.scene.shape.Line
javafx.scene.shape.Shape
   setFill
   setStroke
javafx.scene.text.Font
javafx.scene.text.Text
   getBoundsInParent
   setFont
javafx.stage.Stage
   setScene
   setTitle
   show
```

REVIEW EXERCISES

■ **R10.1** What is the difference between a stage and a pane?

■ **R10.2** From a programmer's perspective, what is the most important difference between the user interface of a console application and a graphical application?

■ **R10.3** What happens when you don't call `setTitle` on the stage object?

■ **R10.4** What happens if you try adding a button directly to a scene?

■ **R10.5** What is an event object? An event source? An event listener?

■ **R10.6** Who calls the `handle` method of an event handler? When does the call to the `handle` method occur?

■ **R10.7** You can exit a graphical program by calling `System.exit(0)`. Describe how to provide an Exit button that functions in the same way as closing the window.

■■ **R10.8** How would you add a counter to the program in Section 10.2.1 that prints how often the button has been clicked? Where is the counter updated?

■■ **R10.9** How would you add a counter to the program in Section 10.2.2 that shows how often the button has been clicked? Where is the counter updated? Where is it displayed?

■■■ **R10.10** How would you reorganize the `InvestmentViewer1` program in Section 10.2.3 if you needed to make the event handler into a class (that is, not a lambda expression)?

■■ **R10.11** Why are we using lambda expressions for event listeners? If Java did not have lambda expressions, could we still implement event listeners? How?

■■ **R10.12** How can you reorganize the `InvestmentViewer1` program so that the root pane is a class that inherits from the `Pane` class?

■ **R10.13** What is the difference between a label, a text field, and a text area?

■■ **R10.14** Name a method that is declared in `TextArea`, a method that `TextArea` inherits from `TextInputControl`, and a method that `TextArea` inherits from `Node`.

■■ **R10.15** How can you rewrite the program in Section 10.3.2 to use a label instead of a text area to show how the interest accumulates? What is the disadvantage of that approach?

■ **R10.16** What happens in the program of Section 10.4.1 if you call `relocate` on the pane containing the bars?

■ **R10.17** In the program of Section 10.4.2, why was the ellipse added before the text?

■■■ **R10.18** How could you center the text precisely within the ellipse in the program of Section 10.4.2?

■■ **R10.19** How would you modify the chart component in Section 10.4.3 to draw a vertical bar chart? (*Hint:* The *y*-values grow downward.)

■ **R10.20** How do you specify a text color?

■■ **R10.21** How do you draw a cat inside a circle?

■ **R10.22** How would you modify the `ItalianFlag` class in How To 10.1 to draw any flag with a white vertical stripe in the middle and two arbitrary colors to the left and right?

PRACTICE EXERCISES

■ **E10.1** Write a program that shows a frame filled with 100 buttons labeled 1 to 100. Nothing needs to happen when you press any of the buttons.

■ **E10.2** Enhance the `ButtonViewer1` program in Section 10.2.1 so that it prints a message "I was clicked *n* times!" whenever the button is clicked. The value *n* should be incremented with each click.

■■ **E10.3** Enhance the `ButtonViewer1` program in Section 10.2.1 so that it has two buttons, each of which prints a message "I was clicked *n* times!" whenever the button is clicked. Each button should have a separate click count.

■■ **E10.4** Enhance the `ButtonViewer1` program in Section 10.2.1 so that it has two buttons labeled A and B, each of which prints a message "Button *x* was clicked!", where *x* is A or B.

■■ **E10.5** Implement a `ButtonViewer1` program as in Exercise •• E10.4 using only a single lambda expression that is generated by a helper method with the button label as parameter.

■ **E10.6** Enhance the `ButtonViewer1` program so that it prints the date and time at which the button was clicked. *Hint:* `System.out.println(new java.util.Date())` prints the current date and time.

■■■ **E10.7** Implement the event handler in the `ButtonViewer2` program of Section 10.2.2 as a regular class (that is, not a lambda expression). *Hint:* Store a reference to the label. Provide a constructor in the handler class that sets the reference.

■■ **E10.8** Add error handling to the program in Section 10.3.2. If the interest rate is not a floating-point number, or if it less than `0`, display an error message.

■ **E10.9** Write a graphical application simulating a bank account. Supply text fields and buttons for depositing and withdrawing money, and for displaying the current balance in a label.

■ **E10.10** Write a graphical application describing an earthquake, as in Section 3.3. Supply a text field and button for entering the strength of the earthquake. Display the earthquake description in a label.

■ **E10.11** Write a graphical application for computing statistics of a data set. Supply a text field and button for adding floating-point values, and display the current minimum, maximum, and average in a label.

■ **E10.12** Write an application with three labeled text fields, one each for the initial amount of a savings account, the annual interest rate, and the number of years. Add a button "Calculate" and a read-only text area to display the balance of the savings account after the end of each year.

■■ **E10.13** In the application from Exercise • E10.12, replace the text area with a bar chart that shows the balance after the end of each year.

■ **E10.14** Write a graphics program that draws your name in red, contained inside a blue rectangle.

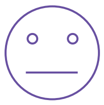
■ **E10.15** Write a graphics program that draws 12 strings, one each for the colors in Table 1, each in its own color. Use a black background for `Color.WHITE`.

■ **E10.16** Write a program that draws two solid squares: one in pink and one in purple. Use a named color for one of them and a custom color for the other.

■■ **E10.17** The `ImageView` constructor accepts a string specifying the image location. If the string starts with `http://`, the image is read from the given URL. Otherwise, the image is read from the file system. If you just specify a file name and no directory, the file must be located in the same directory as the class file of the program. Write a test program that demonstrates reading an image from the web, an image from the default directory, and an image from the download directory of your computer.

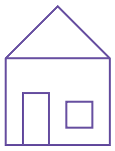
PROGRAMMING PROJECTS

■ **P10.1** Write a program to plot the following face.



•• P10.2 Draw a "bull's eye"—a set of concentric rings in alternating black and white colors. *Hint:* Fill a black circle, then fill a smaller white circle on top, and so on.

•• P10.3 Write a program that draws a picture of a house. It could be as simple as the accompanying figure, or if you like, make it more elaborate (3-D, skyscraper, marble columns in the entryway, whatever).

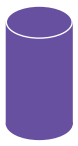


•• P10.4 Extend Exercise •• P10.3 by supplying a `House` class, whose constructor allows you to specify the size. Then populate your frame with a few houses of different sizes.

•• P10.5 Extend Exercise •• P10.4 so that you can make the houses appear in different colors.

• P10.6 Improve the output quality of the investment application in Section 10.3.2. Format the numbers to all have the same width. Set the font of the text area to a fixed width font (such as Courier New or Monospace).

•• P10.7 Write a program that draws a 3D view of a cylinder.

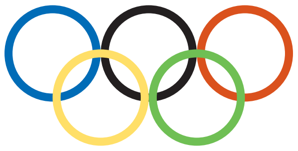


•• P10.8 Write a program to plot the string "HELLO", using only lines and circles. Do not use `Text`, `Label`, or `System.out`.

•• P10.9 Enhance the `InvestmentViewer2` program of Section 10.3.1 so that invalid inputs are detected and the user is notified. Catch the exception thrown by `Double.parseDouble` and place an error message next to the text field. Be sure to clear the error message when the user provides new input.

•• P10.10 Modify the program in How To 10.1 to draw any flag with three horizontal colored stripes. Write a program that displays the German and Hungarian flags.

•• P10.11 Write a program that displays the Olympic rings. Color the rings in the Olympic colors.



■■ P10.12 Write a program that prompts the user to enter an integer in a text field. When a Draw button is clicked, draw as many rectangles at random positions in a pane as the user requested.

■■ P10.13 Write a program that asks the user to enter an integer *n* into a text field. When a Draw button is clicked, draw an *n*-by-*n* grid.

■■■ P10.14 Write a program that has a Draw button and a pane in which a random mixture of rectangles, ellipses, and lines, with random positions, is displayed each time the Draw button is clicked.

■■ P10.15 Make a bar chart to plot the following data set. Label each bar.

Bridge Name	Longest Span (ft)
Golden Gate	4,200
Brooklyn	1,595
Delaware Memorial	2,150
Mackinac	3,800

■■■ P10.16 Write a program that draws a clock face with a time that the user enters in two text fields (one for the hours, one for the minutes).

Hint: You need to determine the angles of the hour hand and the minute hand. The angle of the minute hand is easy; the minute hand travels 360 degrees in 60 minutes. The angle of the hour hand is harder; it travels 360 degrees in 12 × 60 minutes.

■■ Business P10.17 Implement a graphical application that simulates a cash register. Provide a text field for the item price and two buttons for adding the item to the sale, one for taxable items and one for nontaxable items. In a text area, display the register tape that lists all items (labeling the taxable items with a *), followed by the amount due. Provide another button for starting a new sale.

■■ Business P10.18 Write a graphical application to implement a currency converter between euros and U.S. dollars, and vice versa. Provide two text fields for the euro and dollar amounts. Between them, place two buttons labeled > and < for updating the field on the right or left. For this exercise, use a conversion rate of 1 euro = 1.12 U.S. dollars.

■■ Business P10.19 Write a graphical application that produces a restaurant bill. Provide buttons for ten popular dishes or drink items. (You decide on the items and their prices.) Provide text fields for entering less popular items and prices. In a text area, show the bill, including tax and a suggested tip.

© Juanmonino/iStockphoto.

ANSWERS TO SELF-CHECK QUESTIONS

1.
```
Pane root = new Pane();
final int PANE_SIZE = 300;
root.setMinSize(PANE_SIZE, PANE_SIZE);
Scene scene1 = new Scene(root);
stage1.setScene(scene1);
stage1.setTitle("Hello, World!");
stage1.show();
```

2. Set up the root pane like this:

```
Button button1 = new Button("Yes");
Button button2 = new Button("No");
button2.relocate(0, 50);
Pane root = new Pane(button1, button2);
```

3. The button and label are both in the top-left corner.

4. The window is just large enough to contain the button and the label.

5. You get a blank window in “landscape” orientation.

6. The `button1` object is the event source. The lambda expression is the event handler.

7. The `ClickHandler` class implements the `EventHandler` interface.

8. You don’t. The FX library calls the method when the button is clicked.

9. The program would not compile because a lambda expression can only access local variables that are effectively final.

10. The program would continue to work because `resultLabel` is not used outside the `createRootPane` method and the button handler, and it is effectively final.

11. Then the text field is not labeled, and the user will not know its purpose.

12. `Integer.parseInt(textField.getText())`

13. A text field holds a single line of text; a text area holds multiple lines.

14. The text area is intended to display the program output. It does not collect user input.

15. Yes, we did just that in the `ButtonViewer2` program. But the `Label` class doesn’t have an `appendText` method, so you have to call `resultLabel.setText(resultLabel.getText() + ...)`.

16. Here is one possible solution:

```
Rectangle square1 = new Rectangle(
   0, 0, 50, 50);
Rectangle square2 = new Rectangle(
   0, 100, 50, 50);
```

17.
```
Circle shape1 = new Circle(100, 100, 25);
shape1.setstroke = (Color.RED);
shape1.setfill = (Color.RED);
```

18.
```
Line shape1 = new Line(0, 0, 10, 30);
Line shape1 = new Line(10, 30, 20, 0);
```

19. `Label message = new Label("V");`

20. `0, 0, 255`