CHAPTER

31

Advanced JavaFX and FXML

Objectives

- To specify styles for UI nodes using JavaFX CSS (§31.2).
- To create quadratic curve, cubic curve, and path using the QuadCurve, CubicCurve, and Path classes (§31.3).
- To translate, rotate, and scale to perform coordinate transformations for nodes (§31.4).
- To define a shape's border using various types of strokes (§31.5).
- To create menus using the Menu, MenuItem, CheckMenuItem, and RadioMemuItem classes (§31.6).
- To create context menus using the ContextMenu class (§31.7).
- To use SplitPane to create adjustable horizontal and vertical panes (§31.8).
- To create tab panes using the TabPane control (§31.9).
- To create and display tables using the TableView and TableColumn classes (§31.10).
- To create JavaFX user interfaces using FMXL and the visual Scene Builder (§31.11).

31.1 Introduction

JavaFX can be used to develop comprehensive rich Internet applications.

Chapters 14–16 introduced basics of JavaFX, event-driven programming, animations, and simple UI controls. This chapter introduces some advanced features for developing comprehensive GUI applications.

31.2 JavaFX CSS

JavaFX cascading style sheets can be used to specify styles for UI nodes.

JavaFX cascading style sheets are based on CSS with some extensions. CSS defines the style for webpages. It separates the contents of webpages from its style. JavaFX CSS can be used to define the style for the UI and separates the contents of the UI from the style. You can define the look and feel of the UI in a JavaFX CSS file and use the style sheet to set the color, font, margin, and border of the UI components. A JavaFX CSS file makes it easy to modify the style without modifying the Java source code.

A JavaFX style property is defined with a prefix `-fx-` to distinquish it from a property in CSS. All the available JavaFX properties are defined in http://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html. Listing 31.1 gives an example of a style sheet.

Listing 31.1 mystyle.css

```
.plaincircle {
  -fx-fill: white;
  -fx-stroke: black;
}
.circleborder {
  -fx-stroke-width: 5;
  -fx-stroke-dash-array: 12 2 4 2;
}
.border {
  -fx-border-color: black;
  -fx-border-width: 5;
}
#redcircle {
  -fx-fill: red;
  -fx-stroke: red;
}
#greencircle {
  -fx-fill: green;
  -fx-stroke: green;
}
```

A style sheet uses the style class or style id to define styles. Multiple style classes can be applied to a single node, and a style id to a unique node. The syntax `.styleclass` defines a style class. Here, the style classes are named `plaincircle`, `circleborder`, and `circleborder`. The syntax `#styleid` defines a style id. Here, the style ids are named `redcircle` and `greencircle`.

Each node in JavaFX has a `styleClass` variable of the `List<String>` type, which can be obtained from invoking `getStyleClass()`. You can add multiple style classes to a node and only one id to a node. Each node in JavaFX has an `id` variable of the `String` type, which can be set using the `setID(String id)` method. You can set only one id to a node.

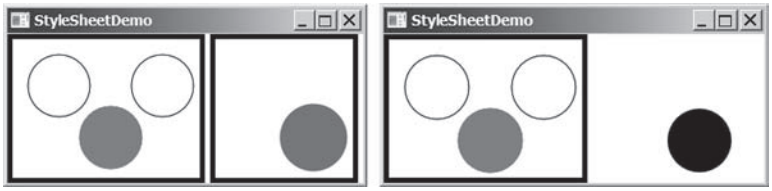
The `Scene` and `Parent` classes have the `stylesheets` property, which can be obtained from invoking the `getStylesheets()` method. This property is of the

ObservableList<String> type. You can add multiple style sheets into this property. You can load a style sheet into a Scene or a Parent. Note that Parent is the superclass for containers and UI control.

Listing 31.2 gives an example that uses the style sheet defined in Listing 31.1.

LISTING 31.2 StyleSheetDemo.java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class StyleSheetDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    HBox hBox = new HBox(5);
    Scene scene = new Scene(hBox, 300, 250);
    scene.getStylesheets().add("mystyle.css"); // Load the stylesheet

    Pane pane1 = new Pane();
    Circle circle1 = new Circle(50, 50, 30);
    Circle circle2 = new Circle(150, 50, 30);
    Circle circle3 = new Circle(100, 100, 30);
    pane1.getChildren().addAll(circle1, circle2, circle3);
    pane1.getStyleClass().add("border");

    circle1.getStyleClass().add("plaincircle"); // Add a style class
    circle2.getStyleClass().add("plaincircle"); // Add a style class
    circle3.setId("redcircle"); // Add a style id

    Pane pane2 = new Pane();
    Circle circle4 = new Circle(100, 100, 30);
    circle4.getStyleClass().addAll("circleborder", "plainCircle");
    circle4.setId("greencircle"); // Add a style class
    pane2.getChildren().add(circle4);
    pane2.getStyleClass().add("border");

    hBox.getChildren().addAll(pane1, pane2);

    primaryStage.setTitle("StyleSheetDemo"); // Set the window title
    primaryStage.setScene(scene); // Place the scene in the window
    primaryStage.show(); // Display the window
  }
}
```



(a) (b)

FIGURE 31.1 The style sheet is used to style the nodes in the scene.

The program loads the style sheet from the file mystyle.css by adding it to the stylesheets property (line 13). The file should be placed in the same directory with the source code for it to run correctly. After the style sheet is loaded, the program sets the style class plaincircle for circle1 and circle2 (lines 22 and 23) and sets the style id redcircle for circle3 (line 24). The program sets style classes circleborder and plaincircle and an id greencircle for circle4 (lines 28 and 29). The style class border is set for both pane1 and pane2 (lines 20 and 31).

The style sheet is set in the scene (line 13). All the nodes inside the scene can use this style sheet. What would happen if line 13 is deleted and the following line is inserted after line 15?

```
pane1.getStylesheets().add("mystyle.css");
```

In this case, only pane1 and the nodes inside pane1 can access the style sheet, but pane2 and circle4 cannot use this style sheet. Therefore, everything in pane1 is displayed the same as before the change, and pane2 and circle4 are displayed without applying the style class and id, as shown in Figure 31.1b.

Note the style class plaincircle and id greencircle both are applied to circle4 (lines 28 and 29). plaincircle sets fill to white and greencircle sets fill to green. The property settings in id take precedence over the ones in classes. Thus, circle4 is displayed in green in this program.

Check Point

31.2.1 How do you load a style sheet to a Scene or a Parent? Can you load multiple style sheets?

31.2.2 If a style sheet is loaded from a node, can the pane and all its containing nodes access the style sheet?

31.2.3 Can a node add multiple style classes? Can a node set multiple style ids?

31.2.4 If the same property is defined in both a style class and a style id and applied to a node, which one has the precedence?

31.3 QuadCurve, CubicCurve, and Path

Key Point

JavaFX provides the QuadCurve, CubicCurve, and Path classes for creating advanced shapes.

Section 14.11 introduced drawing simple shapes using the Line, Rectangle, Circle, Ellipse, Arc, Polygon, and Polyline classes. This section introduces drawing advanced shapes using the CubicCurve, QuadCurve, and Path classes.

31.3.1 QuadCurve and CubicCurve

JavaFX provides the QuadCurve and CubicCurve classes for modeling quadratic curves and cubic curves. A quadratic curve is mathematically defined as a quadratic polynomial. To create a QuadCurve, use its no-arg constructor or the following constructor:

```
QuadCurve(double startX, double startY,
  double controlX, double controlY, double endX, double endY)
```

where (startX, startY) and (endX, endY) specify two endpoints and (controlX, controlY) is a control point. The control point is usually not on the curve instead of defining the trend of the curve, as shown in Figure 31.2a. Figure 31.3 shows the UML diagram for the QuadCurve class.

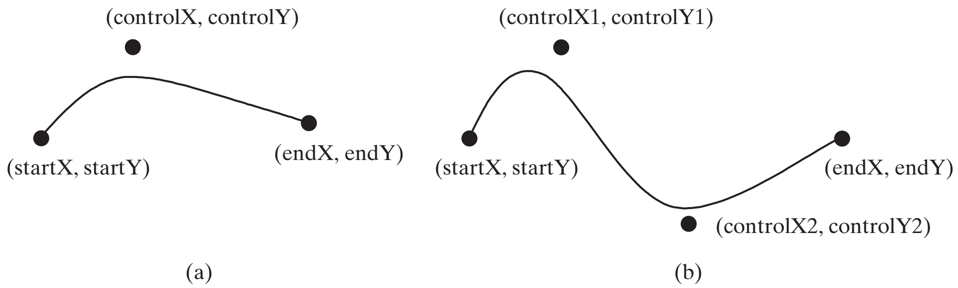
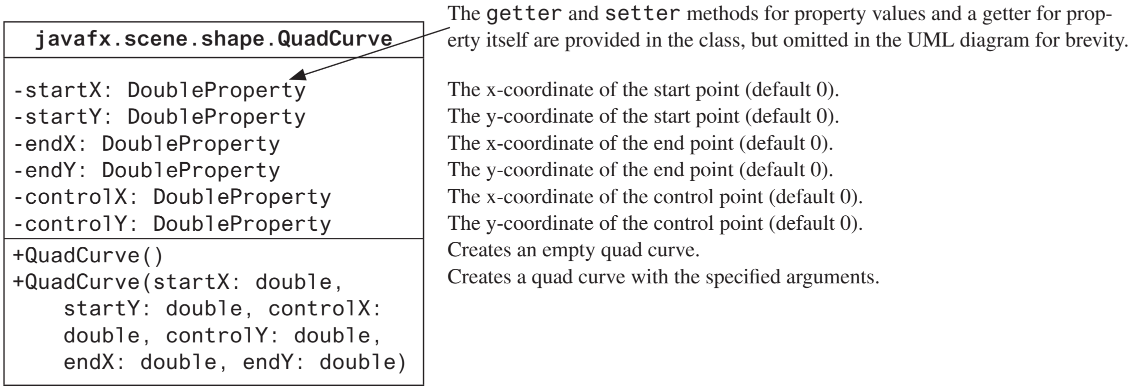


Figure 31.2 (a) A quadratic curve is specified using three points. (b) A cubic curve is specified using four points.



```
javafx.scene.shape.QuadCurve

-startX: DoubleProperty
-startY: DoubleProperty
-endX: DoubleProperty
-endY: DoubleProperty
-controlX: DoubleProperty
-controlY: DoubleProperty

+QuadCurve()
+QuadCurve(startX: double,
   startY: double, controlX:
   double, controlY: double,
   endX: double, endY: double)
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

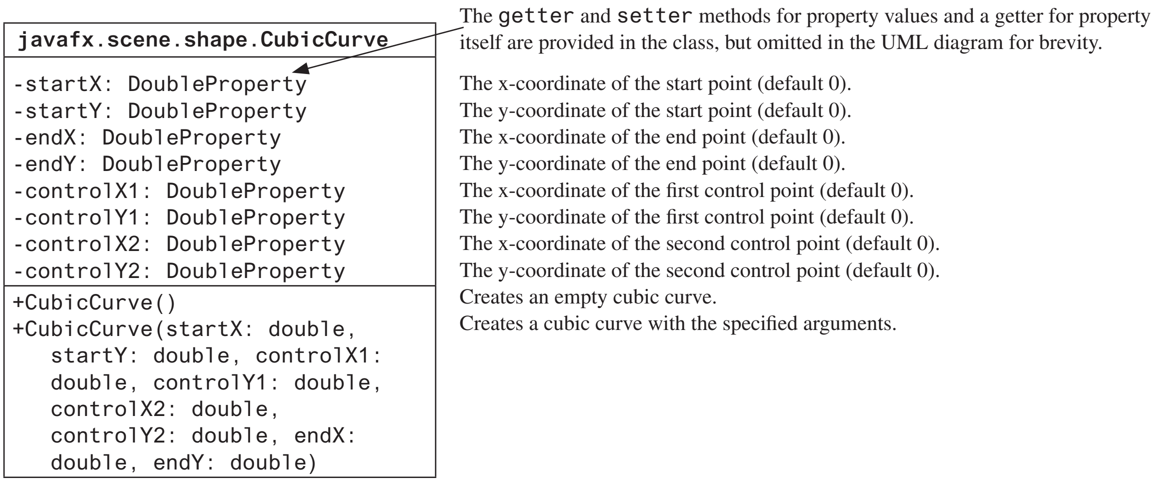
The x-coordinate of the start point (default 0).
The y-coordinate of the start point (default 0).
The x-coordinate of the end point (default 0).
The y-coordinate of the end point (default 0).
The x-coordinate of the control point (default 0).
The y-coordinate of the control point (default 0).
Creates an empty quad curve.
Creates a quad curve with the specified arguments.

Figure 31.3 QuadCurve defines a quadratic curve.

A cubic curve is mathematically defined as a cubic polynomial. To create a CubicCurve, use its no-arg constructor or the following constructor:

```
CubicCurve(double startX, double startY, double controlX1,
  double controlY1, double controlX2, double controlY2,
  double endX, double endY)
```

where (startX, startY) and (endX, endY) specify two endpoints and (controlX1, controlY1) and (controlX2, controlY2) are two control points. The control points are usually not on the curve, instead define the trend of the curve, as shown in Figure 31.2b. Figure 31.4 shows the UML diagram for the CubicCurve class.



```
javafx.scene.shape.CubicCurve

-startX: DoubleProperty
-startY: DoubleProperty
-endX: DoubleProperty
-endY: DoubleProperty
-controlX1: DoubleProperty
-controlY1: DoubleProperty
-controlX2: DoubleProperty
-controlY2: DoubleProperty

+CubicCurve()
+CubicCurve(startX: double,
  startY: double, controlX1:
  double, controlY1: double,
  controlX2: double,
  controlY2: double, endX:
  double, endY: double)
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

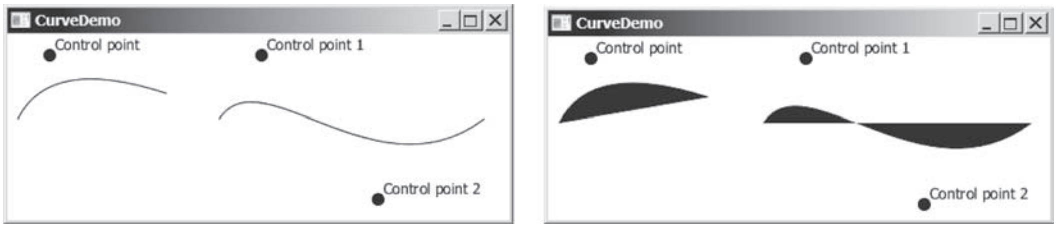
The x-coordinate of the start point (default 0).
The y-coordinate of the start point (default 0).
The x-coordinate of the end point (default 0).
The y-coordinate of the end point (default 0).
The x-coordinate of the first control point (default 0).
The y-coordinate of the first control point (default 0).
The x-coordinate of the second control point (default 0).
The y-coordinate of the second control point (default 0).
Creates an empty cubic curve.
Creates a cubic curve with the specified arguments.

Figure 31.4 CubicCurve defines a quadratic curve.

Listing 31.3 gives a program that demonstrates how to draw quadratic and cubic curves. Figure 31.5a shows a sample run of the program.

LISTING 31.3 CurveDemo.java

```
 1  import javafx.application.Application;
 2  import javafx.scene.Scene;
 3  import javafx.scene.layout.Pane;
 4  import javafx.scene.text.Text;
 5  import javafx.scene.shape.Circle;
 6  import javafx.scene.paint.Color;
 7  import javafx.scene.shape.*;
 8  import javafx.stage.Stage;
 9
10  public class CurveDemo extends Application {
11    @Override // Override the start method in the Application class
12    public void start(Stage primaryStage) {
13      Pane pane = new Pane();
14
15      // Create a QuadCurve
16      QuadCurve quadCurve = new QuadCurve(10, 80, 40, 20, 150, 56);
17      quadCurve.setFill(Color.WHITE);
18      quadCurve.setStroke(Color.BLACK);
19
20      pane.getChildren().addAll(quadCurve, new Circle(40, 20, 6),
21        new Text(40 + 5, 20 - 5, "Control point"));
22
23      // Create a CubicCurve
24      CubicCurve cubicCurve = new CubicCurve
25        (200, 80, 240, 20, 350, 156, 450, 80);
26      cubicCurve.setFill(Color.WHITE);
27      cubicCurve.setStroke(Color.BLACK);
28
29      pane.getChildren().addAll(cubicCurve, new Circle(240, 20, 6),
30        new Text(240 + 5, 20 – 5, "Control point 1"),
31        new Circle(350, 156, 6),
32        new Text(350 + 5, 156 – 5, "Control point 2"));
33
34      Scene scene = new Scene(pane, 300, 250);
35      primaryStage.setTitle("CurveDemo"); // Set the window title
36      primaryStage.setScene(scene); // Place the scene in the window
37      primaryStage.show(); // Display the window
38    }
39  }
```



(a) (b)

FIGURE 31.5 You can draw quadratic and cubic curves using QuadCurve and CubicCurve.

The program creates a QuadCurve with the specified start, control, and end points (line 16) and places the QuadCurve to the pane (line 20). To illustrate the control point, the program also displays the control point as a solid circle (line 21).

The program creates a CubicCurve with the specified start, first control, second control, and end points (lines 24 and 25) and places the CubicCurve to the pane (line 29). To illustrate the control points, the program also displays the control points in the pane (lines 29–32).

Note the curves are filled with color. The program sets the color to white and stroke to black in order to display the curves (lines 17 and 18, 26 and 27). If these code lines are removed from the program, the sample run would look like the one in Figure 31.5b.

31.3.2 Path

The Path class models an arbitrary geometric path. A path is constructed by adding path elements into the path. The PathElement is the root class for the path elements MoveTo, HLineTo, VLineTo, LineTo, ArcTo, QuadCurveTo, CubicCurveTo, and ClosePath.

You can create a Path using its no-arg constructor. The process of the path construction can be viewed as drawing with a pen. The path does not have a default initial position. You need to set an initial position by adding a MoveTo(startX, startY) path element to the path. Adding a HLineTo(newX) element draws a horizontal line from the current position to the new *x*-coordinate. Adding a VLineTo(newY) element draws a vertical line from the current position to the new *y*-coordinate. Adding a LineTo(newX, newY) element draws a line from the current position to the new position. Adding an ArcTo(radiusX, radiusY, xAxisRotation, newX, newY, largeArcFlag, sweepArcFlag) element draws an arc from the previous position to the new position with the specified radius. Adding a QuadCurveTo(controlX, controlY, newX, newY) element draws a quadratic curve from the previous position to the new position with the specified control point. Adding a CubicCurveTo(controlX1, controlY1, controlX2, controlY2, newX, newY) element draws a cubic curve from the previous position to the new position with the specified control points. Adding a ClosePath() element closes the path by drawing a line that connects the starting point with the end point of the path.

Listing 31.4 gives an example that creates a path. A sample run of the program is shown in Figure 31.6.

Listing 31.4 PathDemo.java

```
 1 import javafx.application.Application;
 2 import javafx.scene.Scene;
 3 import javafx.scene.layout.Pane;
 4 import javafx.scene.paint.Color;
 5 import javafx.scene.shape.*;
 6 import javafx.stage.Stage;
 7
 8 public class PathDemo extends Application {
 9   @Override // Override the start method in the Application class
10   public void start(Stage primaryStage) {
11     Pane pane = new Pane();
12
13     // Create a Path
14     Path path = new Path();
15     path.getElements().add(new MoveTo(50.0, 50.0));
16     path.getElements().add(new HLineTo(150.5));
17     path.getElements().add(new VLineTo(100.5));
18     path.getElements().add(new LineTo(200.5, 150.5));
19
20     ArcTo arcTo = new ArcTo(45, 45, 250, 100.5,
21       false, true);
22     path.getElements().add(arcTo);
23
24     path.getElements().add(new QuadCurveTo(50, 50, 350, 100));
25     path.getElements().add(
26       new CubicCurveTo(250, 100, 350, 250, 450, 10));
27
28      path.getElements().add(new ClosePath());
29
30      pane.getChildren().add(path);
31      path.setFill(null);
32      Scene scene = new Scene(pane, 300, 250);
33      primaryStage.setTitle("PathDemo"); // Set the window title
34      primaryStage.setScene(scene); // Place the scene in the window
35      primaryStage.show(); // Display the window
36    }
37  }
```

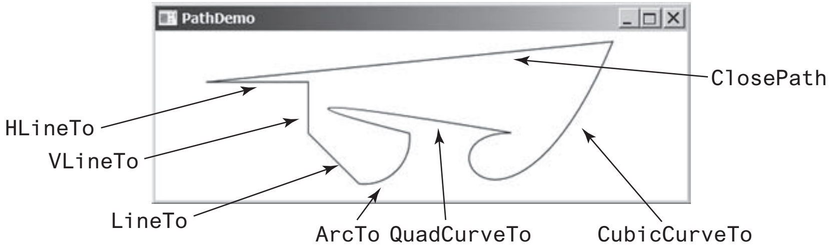


FIGURE 31.6 You can draw a path by adding path elements.

The program creates a `Path` (line 14), moves its position (line 15), and adds a horizontal line (line 16), a vertical line (line 17), and a line (line 18). The `getElements()` method returns an `ObservableList<PathElement>`.

The program creates an `ArcTo` object (lines 20 and 21). The `ArcTo` class contains the `largeArcFlag` and `sweepFlag` properties. By default, these property values are `false`. You may set these properties to `ture` to display a large arc in the opposite direction.

The program adds a quadratic curve (line 24) and a cubic curve (lines 25 and 26) and closes the path (line 28).

By default, the path is not filled. You may change the `fill` property in the path to specify a color to fill the path.

Check Point

31.3.1 Create a `QuadCurve` with starting point (100, 75.5), control point (40, 55.5), and end point (56, 80). Set its `fill` property to white and `stroke` to green.

31.3.2 Create `CubicCurve` object with starting point (100, 75.5), control point 1 (40, 55.5), control point 2 (78.5, 25.5), and end point (56, 80). Set its `fill` property to white and `stroke` to green.

31.3.3 Does a path have a default initial position? How do you set a position for a path?

31.3.4 How do you close a path?

31.3.5 How do you display a filled path?

31.4 Coordinate Transformations

Key Point

JavaFX supports coordinate transformations using translation, rotation, and scaling.

You have used the `rotate` method to rotate a node. You can also perform translations and scaling.

31.4.1 Translations

You can use the `setTranslateX(double x)`, `setTranslateY(double y)`, and `setTranslateZ(double z)` methods in the `Node` class to translate the coordinates for a node. For example, setTranslateX(5) moves the node 5 pixels to the right and setTranslateY(-10) 10 pixels up from the previous position. Figure 31.7 shows a rectangle displayed before and after applying translation. After invoking rectangle.setTranslateX(-6) and rectangle.setTranslateY(4), the rectangle is moved 6 pixels to the left and 4 pixels down from the previous position. Note the coordinate transformation using translation, rotation, and scaling does not change the contents of the shape being transferred. For example, if a rectangle's *x* is 30 and width is 100, after applying transformations to the rectangle, its *x* is still 30 and width is still 100.

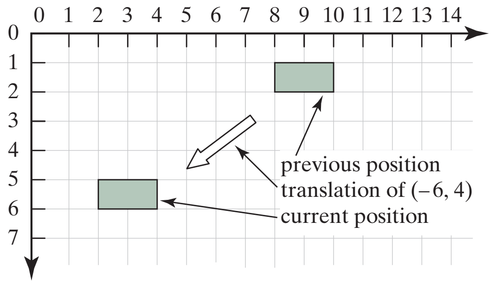


FIGURE 31.7 After applying translation of (−6, 4), the rectangle is moved by the specified distance relative to the previous position.

LISTING 31.5 TranslationDemo.java

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

public class TranslationDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    Pane pane = new Pane();

    double x = 10;
    double y = 10;
    java.util.Random random = new java.util.Random();
    for (int i = 0; i < 10; i++) {
      Rectangle rectangle = new Rectangle(10, 10, 50, 60);
      rectangle.setFill(Color.WHITE);
      rectangle.setStroke(Color.color(random.nextDouble(),
        random.nextDouble(), random.nextDouble()));
      rectangle.setTranslateX(x += 20);
      rectangle.setTranslateY(y += 5);
      pane.getChildren().add(rectangle);
    }

    Scene scene = new Scene(pane, 300, 250);
    primaryStage.setTitle("TranslationDemo"); // Set the window title
    primaryStage.setScene(scene); // Place the scene in the window
    primaryStage.show(); // Display the window
  }
}
```

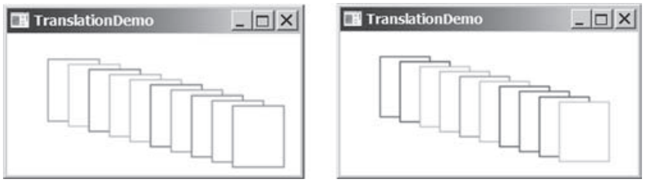


FIGURE 31.8 The rectangles are displayed successively in new locations.

The program repeatedly creates 10 rectangles (line 17). For each rectangle, it sets its `fill` property to white (line 18) and its `stroke` property to a random color (lines 19 and 20), and translates it to a new location (lines 21 and 22). The variables `x` and `y` are used to set the `translateX` and `translateY` properties. These two variable values are changed every time it is applied to a rectangle (see Figure 31.8).

31.4.2 Rotations

Rotation was introduced in Chapter 14. This section discusses it in more depth. You can use the `rotate(double theta)` method in the `Node` class to rotate a node by `theta` degrees from its pivot point clockwise, where `theta` is a double value in degrees. The pivot point is automatically computed based on the bounds of the node. For a circle, ellipse, and a rectangle, the pivot point is the center point of these nodes. For example, `rectangle.rotate(45)` rotates the rectangle 45 degrees clockwise along the eastern direction from the center, as shown in Figure 31.9.

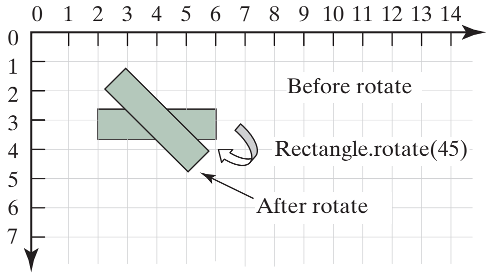


FIGURE 31.9 After performing `rectangle.rotate(45)`, the rectangle is rotated in 45 degrees from the center.

Listing 31.6 gives a program that demonstrates the effect of rotation of coordinates. Figure 31.10 shows a sample run of the program.

LISTING 31.6 `RotateDemo.java`

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

public class RotateDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    Pane pane = new Pane();
    java.util.Random random = new java.util.Random();
13      // The radius of the circle for anchoring rectangles
14      double radius = 90;
15      double width = 20; // Width of the rectangle
16      double height = 40; // Height of the rectangle
17      for (int i = 0; i < 8; i++) {
18        // Center of a rectangle
19        double x = 150 + radius * Math.cos(i * 2 * Math.PI / 8);
20        double y = 150 + radius * Math.sin(i * 2 * Math.PI / 8);
21        Rectangle rectangle = new Rectangle(
22          x - width / 2, y - height / 2, width, height);
23        rectangle.setFill(Color.WHITE);
24        rectangle.setStroke(Color.color(random.nextDouble(),
25          random.nextDouble(), random.nextDouble()));
26        rectangle.setRotate(i * 360 / 8); // Rotate the rectangle
27        pane.getChildren().add(rectangle);
28      }
29
30      Scene scene = new Scene(pane, 300, 300);
31      primaryStage.setTitle("RotateDemo"); // Set the window title
32      primaryStage.setScene(scene); // Place the scene in the window
33      primaryStage.show(); // Display the window
34    }
35  }
```

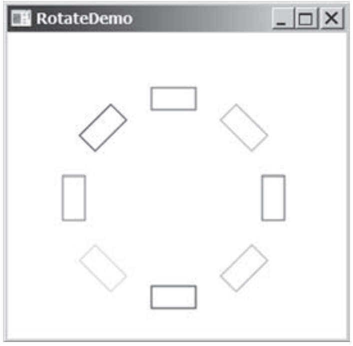


FIGURE 31.10 The `rotate` method rotates a node.

The program creates eight rectangles in a loop (lines 17–28). The center of each rectangle is located on the circle centered as (150, 150) (lines 19 and 20). A rectangle is created by specifying its upper left corner position with width and height (lines 21 and 22). The rectangle is rotated in line 26 and added to the pane in line 27.

31.4.3 Scaling

You can use the `setScaleX(double sx)`, `setScaleY(double sy)`, and `setScaleY(double sy)` methods in the `Node` class to specify a scaling factor. The node will appear larger or smaller depending on the scaling factor. Scaling alters the coordinate space of the node such that each unit of distance along the axis is multiplied by the scale factor. As with rotation transformations, scaling transformations are applied to enlarge or shrink the node around the pivot point. For a node of the rectangle shape, the pivot point is the center of the rectangle. For example, if you apply a scaling factor ($x = 2$, $y = 2$), the entire rectangle including the stroke will double in size, growing to the left, right, up, and down from the center, as shown in Figure 31.11.

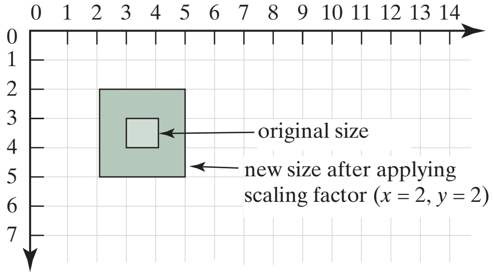


FIGURE 31.11 After applying scaling (*x* = 2, *y* = 2), the node is doubled in size.

Listing 31.7 gives a program that demonstrates the effect of using scaling. Figure 31.12 shows a sample run of the program.

LISTING 31.7 ScaleDemo.java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.shape.Line;
import javafx.scene.text.Text;
import javafx.scene.shape.Polyline;
import javafx.stage.Stage;

public class ScaleDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a polyline to draw a sine curve
    Polyline polyline = new Polyline();
    for (double angle = -360; angle <= 360; angle++) {
      polyline.getPoints().addAll(
        angle, Math.sin(Math.toRadians(angle)));
    }
    polyline.setTranslateY(100);
    polyline.setTranslateX(200);
    polyline.setScaleX(0.5);
    polyline.setScaleY(50);
    polyline.setStrokeWidth(1.0 / 25);

    // Draw x-axis
    Line line1 = new Line(10, 100, 420, 100);
    Line line2 = new Line(420, 100, 400, 90);
    Line line3 = new Line(420, 100, 400, 110);

    // Draw y-axis
    Line line4 = new Line(200, 10, 200, 200);
    Line line5 = new Line(200, 10, 190, 30);
    Line line6 = new Line(200, 10, 210, 30);

    // Draw x, y axis labels
    Text text1 = new Text(380, 70, "X");
    Text text2 = new Text(220, 20, "Y");

    // Add nodes to a pane
    Pane pane = new Pane();
    pane.getChildren().addAll(polyline, line1, line2, line3, line4,
      line5, line6, text1, text2);

43      Scene scene = new Scene(pane, 450, 200);
44      primaryStage.setTitle("ScaleDemo"); // Set the window title
45      primaryStage.setScene(scene); // Place the scene in the window
46      primaryStage.show(); // Display the window
47    }
48  }
```

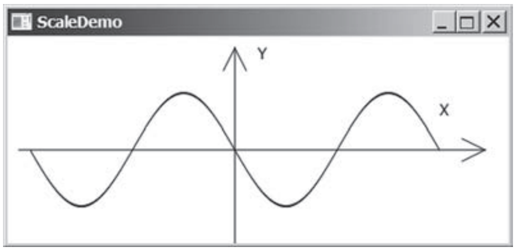


FIGURE 31.12 The `scale` method scales the coordinates in the node.

The program creates a polyline (line 13) and adds the points for a sine curve into the polyline (lines 14–17). Since $|\sin(x)| <= 1$, the y-coordinates are too small. To see the sine curve, the program scales the y-coordinates up by 50 times (line 21) and shrinks the x-coordinates by half (line 20).

Note scaling also causes the stroke width to change. To compensate it, the stroke width is purposely set to 1.0 / 25 (line 22).

31.4.1 Can you perform a coordinate transformation on any node? Does a coordinate transformation change the contents of a `Shape` object?

31.4.2 Does the method `setTranslateX(6)` move the node's x-coordinate to 6? Does the method `setTranslateX(6)` move the node's x-coordinate 6 pixel right from its current location?

31.4.3 Does the method `rotate(Math.PI / 2)` rotate a node 90 degrees? Does the method `rotate(90)` rotate a node 90 degrees?

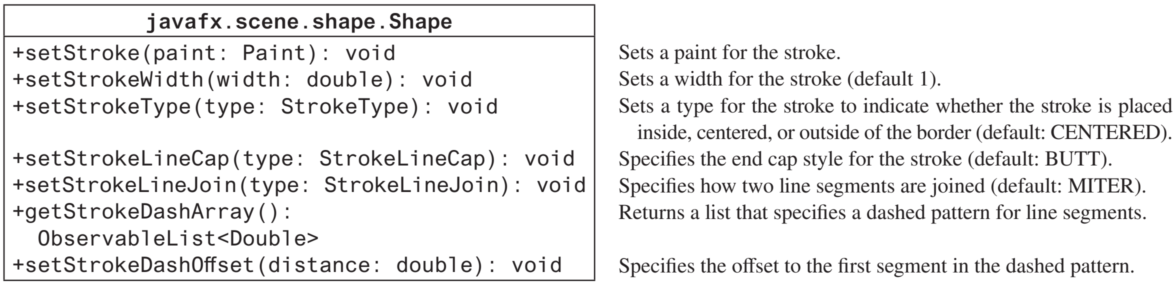
31.4.4 How is the pivot point determined for performing a rotation?

31.4.5 What method do you use to scale a node two times on its x-axis?

31.5 Strokes

Stroke defines a shape's border line style.

JavaFX allows you to specify the attributes of a shape's boundary using the methods in Figure 31.13.



```
javafx.scene.shape.Shape
+setStroke(paint: Paint): void
+setStrokeWidth(width: double): void
+setStrokeType(type: StrokeType): void

+setStrokeLineCap(type: StrokeLineCap): void
+setStrokeLineJoin(type: StrokeLineJoin): void
+getStrokeDashArray():
  ObservableList<Double>
+setStrokeDashOffset(distance: double): void
```

Sets a paint for the stroke.
Sets a width for the stroke (default 1).
Sets a type for the stroke to indicate whether the stroke is placed inside, centered, or outside of the border (default: CENTERED).
Specifies the end cap style for the stroke (default: BUTT).
Specifies how two line segments are joined (default: MITER).
Returns a list that specifies a dashed pattern for line segments.
Specifies the offset to the first segment in the dashed pattern.

FIGURE 31.13 The `Shape` class contains the methods for setting stroke properties.

The `setStroke(paint)` method sets a paint for the stroke. The width of the stroke can be specified using the `setStrokeWidth(width)` method.

The `setStrokeType(type)` method sets a type for the stroke. The type defines whether the stroke is inside, outside, or in the center of the border using the constants `StrokeType.INSIDE`, `StrokeType.OUTSIDE`, or `StrokeType.CENTERED` (default), as shown in Figure 31.14.

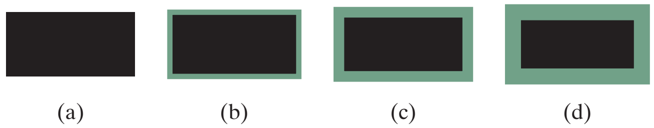


Figure 31.14 (a) No stroke is used. (b) A stroke is placed inside the border. (c) A stroke is placed in the center of the border. (d) A stroke is placed outside of the border.

Note for the centered style, the stroke is applied by extending the boundary of the node by a distance of half of the `strokeWidth` on either side (inside and outside) of the boundary.

The `setStrokeLineCap(capType)` method sets an end cap style for the stroke. The styles are defined as `StrokeLineCap.BUTT` (default), `StrokeLineCap.ROUND`, and `StrokeLineCap.SQUARE`, as illustrated in Figure 31.15. The `BUTT` stroke ends an unclosed path with no added decoration. The `ROUND` stroke ends an unclosed side of a path with an added half circle whose radius is half of the stroke width. The `SQUARE` stroke ends an unclosed side of a path with an added square that extends half of the stroke width.

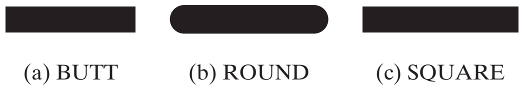


Figure 31.15 (a) No decoration for a BUTT line cap. (b) A half circle is added to an unclosed path. (c) A square with half of the stroke width is extended to an unclosed path.

The `setStrokeLineJoin` method defines the decoration applied where path segments meet. You can specify three types of line join using the constants `StrokeLineJoin.MITER` (default), `StrokeLineJoin.BEVEL`, and `StrokeLineJoin.ROUND`, as shown in Figure 31.16.

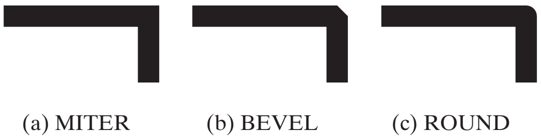


Figure 31.16 Path segments can be joined in three ways: (a) MITER, (b) BEVEL, and (c) ROUND.

The `Shape` class has a property named `strokeDashArray` of the `ObservableList<Double>` type. This property is used to define a dashed pattern for the stroke. Alternate numbers in the list specify the lengths of the opaque and transparent segments of the dashes. For example, the list `[10.0, 20.0, 30.0, 40.0]` specifies a pattern as shown in Figure 31.17.

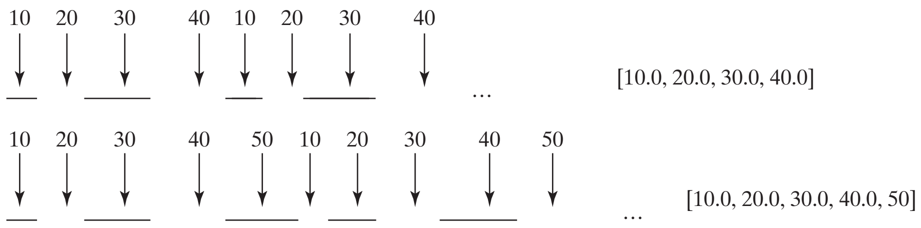


Figure 31.17 The numbers in the list specify the opaque and transparent segments of the stroke alternately.

The `setStrokeDashOffset(distance)` method defines the offset to the first segment in the dash pattern. Figure 31.18 illustrates the offset 5 for the dash list `[10.0, 20.0, 30.0, 40.0]`.

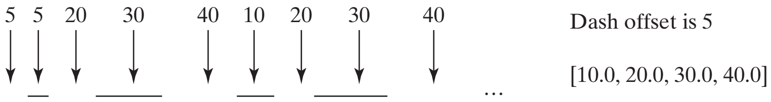


FIGURE 31.18 The dash offset specifies on offset for the first segment.

Listing 31.8 gives a program that demonstrates the methods to set attributes for a stroke. Figure 31.19 shows a sample run of the program.

LISTING 31.8 StrokeDemo.java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.shape.Rectangle;
import javafx.scene.shape.*;

public class StrokeDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    Rectangle rectangle1 = new Rectangle(20, 20, 70, 120);
    rectangle1.setFill(Color.WHITE);
    rectangle1.setStrokeWidth(15);
    rectangle1.setStroke(Color.ORANGE);

    Rectangle rectangle2 = new Rectangle(20, 20, 70, 120);
    rectangle2.setFill(Color.WHITE);
    rectangle2.setStrokeWidth(15);
    rectangle2.setStroke(Color.ORANGE);
    rectangle2.setTranslateX(100);
    rectangle2.setStrokeLineJoin(StrokeLineJoin.BEVEL);

    Rectangle rectangle3 = new Rectangle(20, 20, 70, 120);
    rectangle3.setFill(Color.WHITE);
    rectangle3.setStrokeWidth(15);
    rectangle3.setStroke(Color.ORANGE);
    rectangle3.setTranslateX(200);
    rectangle3.setStrokeLineJoin(StrokeLineJoin.ROUND);

    Line line1 = new Line(320, 20, 420, 20);
    line1.setStrokeLineCap(StrokeLineCap.BUTT);
    line1.setStrokeWidth(20);

    Line line2 = new Line(320, 70, 420, 70);
    line2.setStrokeLineCap(StrokeLineCap.ROUND);
    line2.setStrokeWidth(20);

    Line line3 = new Line(320, 120, 420, 120);
    line3.setStrokeLineCap(StrokeLineCap.SQUARE);
    line3.setStrokeWidth(20);

    Line line4 = new Line(460, 20, 560, 120);
    line4.getStrokeDashArray().addAll(10.0, 20.0, 30.0, 40.0);

46       Pane pane = new Pane();
47       pane.getChildren().addAll(rectangle1, rectangle2, rectangle3,
48         line1, line2, line3, line4);
49
50       Scene scene = new Scene(pane, 610, 180);
51       primaryStage.setTitle("StrokeDemo"); // Set the window title
52       primaryStage.setScene(scene); // Place the scene in the window
53       primaryStage.show(); // Display the window
54     }
55
56     // Launch the program from command-line
57     public static void main(String[] args) {
58       launch(args);
59     }
60   }
```

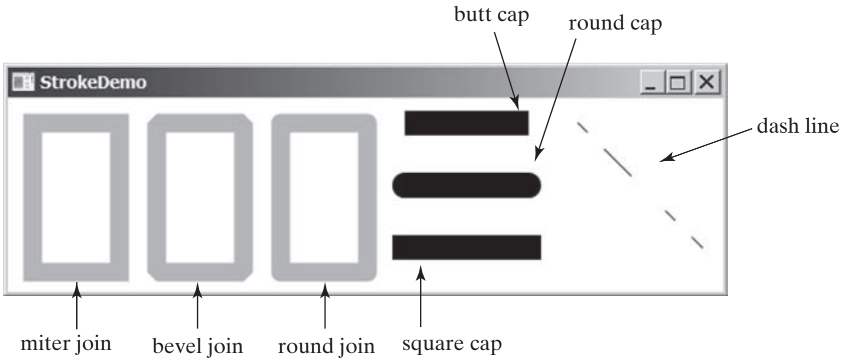


FIGURE 31.19 You can specify the attributes for strokes.

The program creates three rectangles (lines 12–29). Rectangle 1 uses default miter join, rectangle 2 uses bevel join (line 22), and rectangle 3 uses round join (line 29).

The program creates three lines with butt, round, and square end cap (lines 31–41).

The program creates a line and sets dash pattern for this line (line 44). Note the `strokeDashArray` property is of the `ObservableList<Double>` type. You have to add `Double` values to the list. Adding a number such as 10 would cause an error.

Check Point

31.5.1 Are the methods for setting a stroke and its attributes defined in the `Node` or `Shape` class?

31.5.2 How do you set a stroke width to `3` pixels?

31.5.3 What are the stroke types? What is the default stroke type? How do you set a stroke type?

31.5.4 What are the stroke line join types? What is the default stroke line join type? How do you set a stroke line join type?

31.5.5 What are the stroke cap types? What is the default stroke cap type? How do you set a stroke cap type?

31.5.6 How do you specify a dashed pattern for strokes?

31.6 Menus

Key Point

You can create menus in JavaFX.

Menus make selection easier and are widely used in window applications. JavaFX provides five classes that implement menus: `MenuBar`, `Menu`, `MenuItem`, `CheckMenuItem`, and `RadioButtonMenuItem`.

MenuBar is a top-level menu component used to hold the menus. A menu consists of menu items that the user can select (or toggle on or off). A menu item can be an instance of MenuItem, CheckMenuItem, or RadioButtonMenuItem. Menu items can be associated with nodes and keyboard accelerators.

31.6.1 Creating Menus

The sequence of implementing menus in JavaFX is as follows:

1. Create a menu bar and add it to a pane. For example, the following code creates a pane and a menu bar, and adds the menu bar to the pane:

```
MenuBar menuBar = new MenuBar();
Pane pane = new Pane();
pane.getChildren().add(menuBar);
```

2. Create menus and add them under the menu bar. For example, the following creates two menus and adds them to a menu bar, as shown in Figure 31.20a:

```
Menu menuFile = new Menu("File");
Menu menuHelp = new Menu("Help");
menuBar.getMenus().addAll(menuFile, menuHelp);
```

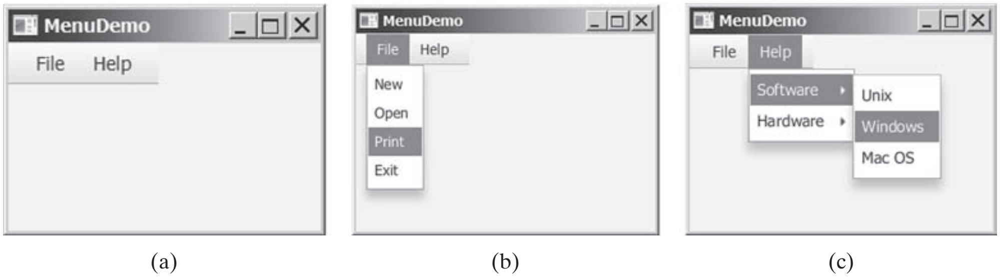


FIGURE 31.20 (a) The menus are placed under a menu bar. (b) Clicking a menu on the menu bar reveals the items under the menu. (c) Clicking a menu item reveals the submenu items under the menu item.

3. Create menu items and add them to the menus.

```
menuFile.getItems().addAll(new MenuItem("New"),
  new MenuItem("Open"), new MenuItem("Print"),
  new MenuItem("Exit"));
```

This code adds the menu items New, Open, Print, and Exit, in this order, to the File menu, as shown in Figure 31.20b.

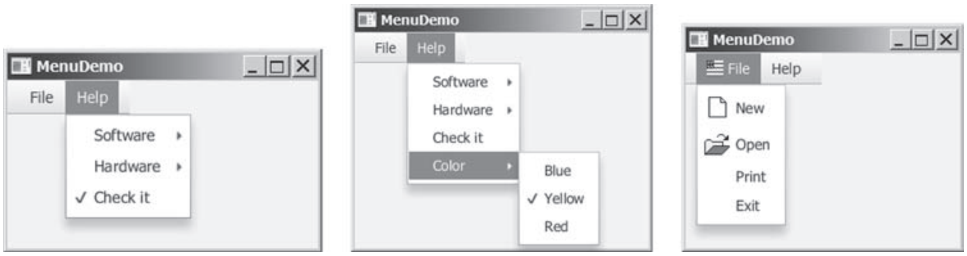
3.1. Creating submenu items.
You can also embed menus inside menus so the embedded menus become submenus. Here is an example:

```
Menu softwareHelpSubMenu = new Menu("Software");
Menu hardwareHelpSubMenu = new Menu("Hardware");
menuHelp.getItems().add(softwareHelpSubMenu);
menuHelp.getItems().add(hardwareHelpSubMenu);
softwareHelpSubMenu.getItems().add(new MenuItem("Unix"));
softwareHelpSubMenu.getItems().add(new MenuItem("Windows"));
softwareHelpSubMenu.getItems().add(new MenuItem("Mac OS"));
```

This code adds two submenus, softwareHelpSubMenu and hardwareHelpSubMenu, in MenuHelp. The menu items Unix, NT, and Win95 are added to softwareHelpSubMenu (see Figure 31.20c).

3.2. Creating check-box menu items.
You can also add a CheckMenuItem to a Menu. CheckMenuItem is a subclass of MenuItem that adds a Boolean state to the MenuItem and displays a check when its state is true. You can click a menu item to turn it on or off. For example, the following statement adds the check-box menu item Check it (see Figure 31.21a).

```
menuHelp.getItems().add(new CheckMenuItem("Check it"));
```



(a) (b) (c)

FIGURE 31.21 (a) A check box menu item lets you check or uncheck a menu item just like a check box. (b) You can use RadioMenuItem to choose among mutually exclusive menu choices. (c) You can set image icons and keyboard accelerators in menus.

3.3. Creating radio menu items.
You can also add radio menu items to a menu, using the RadioMenuItem class. This is often useful when you have a group of mutually exclusive choices in the menu. For example, the following statements add a submenu named Color and a set of radio buttons for choosing a color (see Figure 31.21b):

```
RadioMenuItem rmiBlue, rmiYellow, rmiRed;
colorHelpSubMenu.getItems().add(rmiBlue =
  new RadioMenuItem("Blue"));
colorHelpSubMenu.getItems().add(rmiYellow =
  new RadioMenuItem("Yellow"));
colorHelpSubMenu.getItems().add(rmiRed =
  new RadioMenuItem("Red"));

ToggleGroup group = new ToggleGroup();
rmiBlue.setToggleGroup(group);
rmiYellow.setToggleGroup(group);
rmiRed.setToggleGroup(group);
```

4. The menu items generate ActionEvent. To handle ActionEvent, implement the setOnAction method.
5. **Image Icons and Keyboard Accelerators**

The Menu, CheckMenuItem, and RadioMenuItem are the subclasses of MenuItem. The MenuItem has a graphic property for specifying a node to be displayed in the menu item. Usually, the graphic is an image view. The classes Menu, MenuItem, CheckMenuItem, and RadioMenuItem have another constructor that you can use to specify a graphic. For example, the following code adds an image to the menu, menu item, check menu item, and radio menu item (see Figure 31.21c).

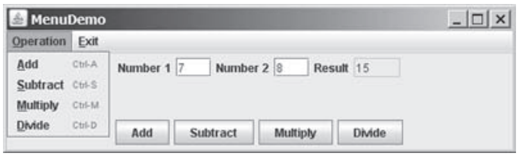
```
Menu menuFile = new Menu("File",
  new ImageView("image/usIcon.gif"));
MenuItem menuItemOpen = new MenuItem("New",
  new ImageView("image/new.gif"));
CheckMenuItem checkMenuItem = new CheckMenuItem("Check it",
  new ImageView("image/us.gif"));
RadioMenuItem rmiBlue = new RadioMenuItem("Blue",
  new ImageView("image/us.gif"));
```

6. A key accelerator lets you select a menu item directly by pressing the CTRL and the accelerator key. For example, by using the following code, you can attach the accelerator key CTRL+N to the Open menu item:

```
menuItemOpen.setAccelerator(
  KeyCombination.keyCombination("Ctrl+O"));
```

31.6.2 Example: Using Menus

This section gives an example that creates a user interface to perform arithmetic. The interface contains labels and text fields for Number 1, Number 2, and Result. The Result text field displays the result of the arithmetic operation between Number 1 and Number 2. Figure 31.22 contains a sample run of the program.



Number 1 7 Number 2 8 Result 15

FIGURE 31.22 Arithmetic operations can be performed by clicking buttons or by choosing menu items from the Operation menu.

Here are the major steps in the program (Listing 31.9):

1. Create a menu bar and add it into a VBox. Create the menus Operation and Exit, and add them to the menu bar. Add the menu items Add, Subtract, Multiply, and Divide under the Operation menu and add the menu item Close under the Exit menu.
2. Create an HBox to hold labels and text fields and place it into the VBox.
3. Create an HBox to hold the four buttons labeled Add, Subtract, Multiply, and Divide and place it into the VBox.
4. Implement the handlers to process the events from the menu items and the buttons.

LISTING 31.9 MenuDemo.java

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuBar;
import javafx.scene.control.MenuItem;
import javafx.scene.control.TextField;
import javafx.scene.input.KeyCombination;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class MenuDemo extends Application {
  private TextField tfNumber1 = new TextField();
  private TextField tfNumber2 = new TextField();
  private TextField tfResult = new TextField();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    MenuBar menuBar = new MenuBar();

    Menu menuOperation = new Menu("Operation");
    Menu menuExit = new Menu("Exit");
    menuBar.getMenus().addAll(menuOperation, menuExit);

    MenuItem menuItemAdd = new MenuItem("Add");
    MenuItem menuItemSubtract = new MenuItem("Subtract");
    MenuItem menuItemMultiply = new MenuItem("Multiply");
    MenuItem menuItemDivide = new MenuItem("Divide");
    menuOperation.getItems().addAll(menuItemAdd, menuItemSubtract,
      menuItemMultiply, menuItemDivide);

    MenuItem menuItemClose = new MenuItem("Close");
    menuExit.getItems().add(menuItemClose);

    menuItemAdd.setAccelerator(
      KeyCombination.keyCombination("Ctrl+A"));
    menuItemSubtract.setAccelerator(
      KeyCombination.keyCombination("Ctrl+S"));
    menuItemMultiply.setAccelerator(
      KeyCombination.keyCombination("Ctrl+M"));
    menuItemDivide.setAccelerator(
      KeyCombination.keyCombination("Ctrl+D"));

    HBox hBox1 = new HBox(5);
    tfNumber1.setPrefColumnCount(2);
    tfNumber2.setPrefColumnCount(2);
    tfResult.setPrefColumnCount(2);
    hBox1.getChildren().addAll(new Label("Number 1:"), tfNumber1,
      new Label("Number 2:"), tfNumber2, new Label("Result:"),
      tfResult);
    hBox1.setAlignment(Pos.CENTER);

    HBox hBox2 = new HBox(5);
    Button btAdd = new Button("Add");
    Button btSubtract = new Button("Subtract");
    Button btMultiply = new Button("Multiply");
    Button btDivide = new Button("Divide");
    hBox2.getChildren().addAll(btAdd, btSubtract, btMultiply, btDivide);
    hBox2.setAlignment(Pos.CENTER);

    VBox vBox = new VBox(10);
    vBox.getChildren().addAll(menuBar, hBox1, hBox2);
    Scene scene = new Scene(vBox, 300, 250);
    primaryStage.setTitle("MenuDemo"); // Set the window title
    primaryStage.setScene(scene); // Place the scene in the window
    primaryStage.show(); // Display the window

```

```
 71      // Handle menu actions
 72      menuItemAdd.setOnAction(e -> perform('+'));
 73      menuItemSubtract.setOnAction(e -> perform('-'));
 74      menuItemMultiply.setOnAction(e -> perform('*'));
 75      menuItemDivide.setOnAction(e -> perform('/'));
 76      menuItemClose.setOnAction(e -> System.exit(0));
 77
 78      // Handle button actions
 79      btAdd.setOnAction(e -> perform('+'));
 80      btSubtract.setOnAction(e -> perform('-'));
 81      btMultiply.setOnAction(e -> perform('*'));
 82      btDivide.setOnAction(e -> perform('/'));
 83    }
 84
 85    private void perform(char operator) {
 86      double number1 = Double.parseDouble(tfNumber1.getText());
 87      double number2 = Double.parseDouble(tfNumber2.getText());
 88
 89      double result = 0;
 90      switch (operator) {
 91        case '+': result = number1 + number2; break;
 92        case '-': result = number1 - number2; break;
 93        case '*': result = number1 * number2; break;
 94        case '/': result = number1 / number2; break;
 95     }
 96
 97     tfResult.setText(result + "");
 98   }
100 }
```

The program creates a menu bar (line 22), which holds two menus: `menuOperation` and `menuExit` (lines 24–36). The `menuOperation` contains four menu items for doing arithmetic: Add, Subtract, Multiply, and Divide. The `menuExit` contains the menu item Close for exiting the program. The menu items in the Operation menu are created with keyboard accelerators (lines 38–45).

The labels and text fields are placed in an `HBox` (lines 47–54) and four buttons are placed in another `HBox` (lines 56–62). The menu bar and these two `HBoxes` are added to a `VBox` (line 65), which is placed in the scene (line 66).

The user enters two numbers in the number fields. When an operation is chosen from the menu, its result, involving two numbers, is displayed in the Result field. The user can also click the buttons to perform the same operation.

The program sets actions for the menu items and buttons in lines 72–82. The private method `perform(char operator)` (lines 85–98) retrieves operands from the text fields in Number 1 and Number 2, applies the binary operator on the operands, and sets the result in the Result text field.

Check Point

31.6.1 How do you create a menu bar, menu, menu item, check menu item, and radio menu item?

31.6.2 How do you place a menu into a menu bar? How do you place a menu item, check menu item, and radio menu item into a menu?

31.6.3 Can you place a menu item into another menu item or a check menu or a radio menu item into a menu item?

31.6.4 How do you associate an image with a menu, menu item, check menu item, and radio menu item?

31.6.5 How do you associate an accelerator CTRL+O with a menu item, check menu item, and radio menu item?

31.7 Context Menus

You can create context menus in JavaFX.

A *context menu*, also known as a *popup menu*, is like a regular menu, but does not have a menu bar and can float anywhere on the screen. Creating a context menu is similar to creating a regular menu. First, you create an instance of `ContextMenu`, and then you can add `MenuItem`, `CheckMenuItem`, and `RadioMenuItem` to the context menu. For example, the following code creates a `ContextMenu`, then adds `MenuItems` into it:

```
ContextMenu contextMenu = new ContextMenu();
ContextMenu.getItems().add(new MenuItem("New"));
ContextMenu.getItems().add(new MenuItem("Open"));
```

A regular menu is always added to a menu bar, but a context menu is associated with a parent node and is displayed using the show method in the `ContextMenu` class. You specify the parent node and the location of the context menu, using the coordinate system of the parent like this:

```
contextMenu.show(node, x, y);
```

Customarily, you display a context menu by pointing to a GUI component and clicking a certain mouse button, the so-called popup trigger. Popup triggers are system dependent. In Windows, the context menu is displayed when the right mouse button is released. In Motif, the context menu is displayed when the third mouse button is pressed and held down.

Listing 31.10 gives an example that creates a pane. When the mouse points to the pane, clicking a mouse button displays a context menu, as shown in Figure 31.23.

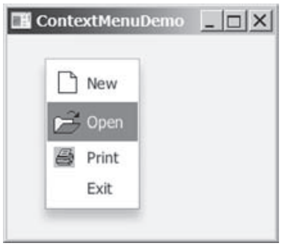


Figure 31.23 A context menu is displayed when the popup trigger is issued on the pane.

Here are the major steps in the program (Listing 31.10):

1. Create a context menu using `ContextMenu`. Create menu items for New, Open, Print, and Exit using `MenuItem`.
2. Add the menu items into the context menu.
3. Create a pane and place it in the scene.
4. Implement the handler to process the events from the menu items.
5. Implement the `mousePressed` handler to display the context menu.

Listing 31.10 ContextMenuDemo.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ContextMenu;
import javafx.scene.control.MenuItem;
import javafx.scene.image.ImageView;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

```

```
 9  public class ContextMenuDemo extends Application {
10    @Override // Override the start method in the Application class
11    public void start(Stage primaryStage) {
12      ContextMenu contextMenu = new ContextMenu();
13      MenuItem menuItemNew = new MenuItem("New",
14        new ImageView("image/new.gif"));
15      MenuItem menuItemOpen = new MenuItem("Open",
16        new ImageView("image/open.gif"));
17      MenuItem menuItemPrint = new MenuItem("Print",
18        new ImageView("image/print.gif"));
19      MenuItem menuItemExit = new MenuItem("Exit");
20      contextMenu.getItems().addAll(menuItemNew, menuItemOpen,
21        menuItemPrint, menuItemExit);
22
23      Pane pane = new Pane();
24      Scene scene = new Scene(pane, 300, 250);
25      primaryStage.setTitle("ContextMenuDemo"); // Set the window title
26      primaryStage.setScene(scene); // Place the scene in the window
27      primaryStage.show(); // Display the window
28
29      pane.setOnMousePressed(
30        e -> contextMenu.show(pane, e.getScreenX(), e.getScreenY()));
31
32      menuItemNew.setOnAction(e -> System.out.println("New"));
33      menuItemOpen.setOnAction(e -> System.out.println("Open"));
34      menuItemPrint.setOnAction(e -> System.out.println("Print"));
35      menuItemExit.setOnAction(e -> System.exit(0));
36    }
37  }
```

The process of creating context menus is similar to the process for creating regular menus. To create a context menu, create a ContextMenu as the basis (line 12) and add MenuItems to it (lines 13–21).

To show a context menu, use the show method by specifying the parent node and the location for the context menu (lines 29 and 30). The show method is invoked when the context menu is triggered by a mouse click on the pane (line 30).

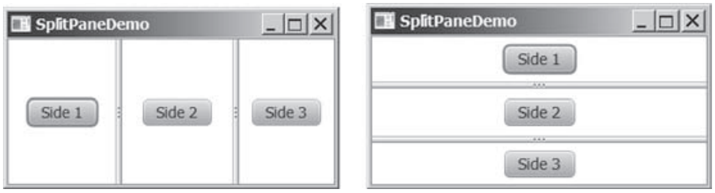
31.7.1 How do you create a context menu? How do you add menu items, check menu items, and radio menu items into a context menu?

31.7.2 How do you show a context menu?

31.8 SplitPane

The SplitPane class can be used to display multiple panes and allow the user to adjust the size of the panes.

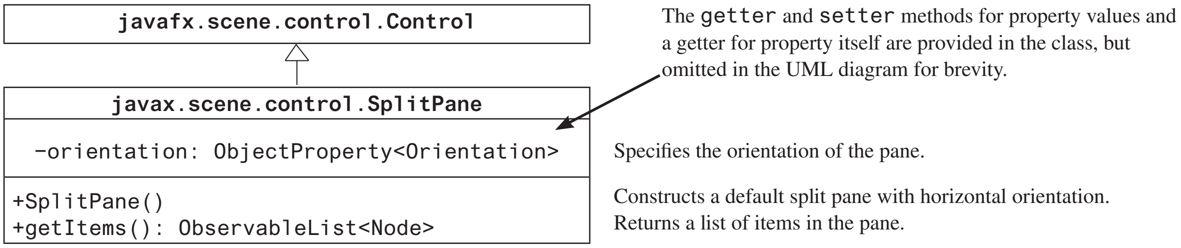
The SplitPane is a control that contains two components with a separate bar known as a divider, as shown in Figure 31.24.



(a) Horizontal orientation (b) Vertical orientation

FIGURE 31.24 SplitPane divides a container into two parts.

The two sides separated by the divider can appear in horizontal or vertical orientation. The divider separating two sides can be dragged to change the amount of space occupied by each side. Figure 31.25 shows the frequently used properties, constructors, and methods in `SplitPane`.


javafx.scene.control.Control
javax.scene.control.SplitPane
-orientation: ObjectProperty<Orientation>
+SplitPane()
+getItems(): ObservableList<Node>

The `getter` and `setter` methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

Specifies the orientation of the pane.

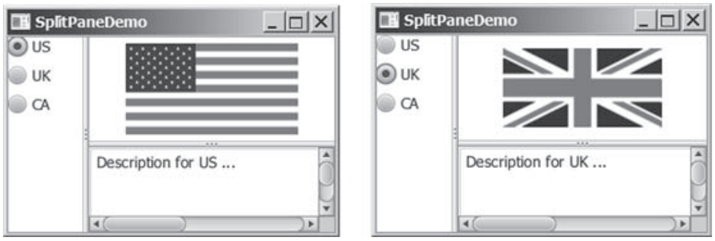
Constructs a default split pane with horizontal orientation.
Returns a list of items in the pane.

FIGURE 31.25 `SplitPane` provides methods to specify the properties of a split pane and for manipulating the components in a split pane.

Listing 31.11 gives an example that uses radio buttons to let the user select a country and displays the country's flag and description in separate sides, as shown in Figure 31.26. The description of the currently selected layout manager is displayed in a text area. The radio buttons, buttons, and text area are placed in two split panes.

LISTING 31.11 `SplitPaneDemo.java`

```java
import javafx.application.Application;
import javafx.geometry.Orientation;
import javafx.scene.Scene;
import javafx.scene.control.RadioButton;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.SplitPane;
import javafx.scene.control.TextArea;
import javafx.scene.control.ToggleGroup;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.StackPane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class SplitPaneDemo extends Application {
  private Image usImage = new Image(
    "http://www.cs.armstrong.edu/liang/common/image/us.gif");
  private Image ukImage = new Image(
    "http://www.cs.armstrong.edu/liang/common/image/uk.gif");
  private Image caImage = new Image(
    "http://www.cs.armstrong.edu/liang/common/image/ca.gif");
  private String usDescription = "Description for US ...";
  private String ukDescription = "Description for UK ...";
  private String caDescription = "Description for CA ...";

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    VBox vBox = new VBox(10);
    RadioButton rbUS = new RadioButton("US");
    RadioButton rbUK = new RadioButton("UK");
    RadioButton rbCA = new RadioButton("CA");
    vBox.getChildren().addAll(rbUS, rbUK, rbCA);

    SplitPane content = new SplitPane();
    content.setOrientation(Orientation.VERTICAL);
    ImageView imageView = new ImageView(usImage);
    StackPane imagePane = new StackPane();
    imagePane.getChildren().add(imageView);
    TextArea taDescription = new TextArea();
    taDescription.setText(usDescription);
    content.getItems().addAll(
      imagePane, new ScrollPane(taDescription));

    SplitPane sp = new SplitPane();
    sp.getItems().addAll(vBox, content);

    Scene scene = new Scene(sp, 300, 250);
    primaryStage.setTitle("SplitPaneDemo"); // Set the window title
    primaryStage.setScene(scene); // Place the scene in the window
    primaryStage.show(); // Display the window

    // Group radio buttons
    ToggleGroup group = new ToggleGroup();
    rbUS.setToggleGroup(group);
    rbUK.setToggleGroup(group);
    rbCA.setToggleGroup(group);

    rbUS.setSelected(true);
    rbUS.setOnAction(e -> {
      imageView.setImage(usImage);
      taDescription.setText(usDescription);
    });

    rbUK.setOnAction(e -> {
      imageView.setImage(ukImage);
      taDescription.setText(ukDescription);
    });

    rbCA.setOnAction(e -> {
      imageView.setImage(caImage);
      taDescription.setText(caDescription);
    });
  }
}
```



Description for US ...

Description for UK ...

FIGURE 31.26 You can adjust the component size in the split panes.

The program places three radio buttons in a VBox (lines 28–32) and creates a vertical split pane for holding an image view and a text area (lines 34–42). Split panes can be embedded. The program creates a horizontal split pane and places the VBox and the vertical split pane into it (lines 44 and 45).

Adding a split pane to an existing split pane results in three split panes. The program creates two split panes (lines 34, 42) to hold a panel for radio buttons, a panel for buttons, and a scroll pane.

The program groups radio buttons (lines 53–56) and processes the action for radio buttons (lines 59–72).

31.8.1 How do you create a horizontal SplitPane? How do you create a vertical SplitPane?

31.8.2 How do you add items into a SplitPane? Can you add a SplitPane to another SplitPane?

31.9 TabPane

The TabPane class can be used to display multiple panes with tabs.

TabPane is a useful control that provides a set of mutually exclusive tabs, as shown in Figure 31.27. You can switch between a group of tabs. Only one tab is visible at a time. A Tab can be added to a TabPane. Tabs in a TabPane can be placed in the position top, left, bottom, or right.

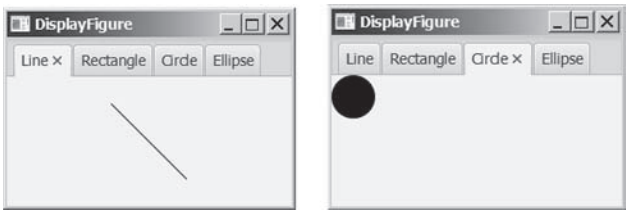
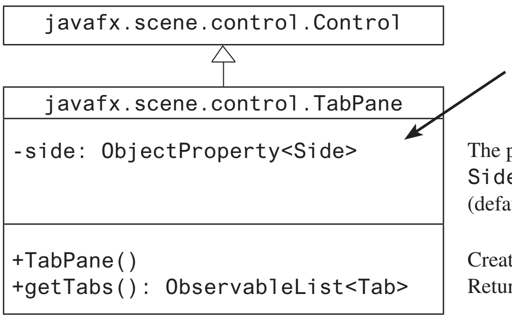


FIGURE 31.27 TabPane holds a group of tabs.

Each tab represents a single page. Tabs are defined in the Tab class. Tabs can contain any Node such as a pane, a shape, or a control. A tab can contain another pane. Therefore, you can create a multilayered tab pane. Figures 31.28 and 31.29 show the frequently used properties, constructors, and methods in TabPane and Tab.


javafx.scene.control.Control
javafx.scene.control.TabPane
-side: ObjectProperty<Side>
+TabPane()
+getTabs(): ObservableList<Tab>

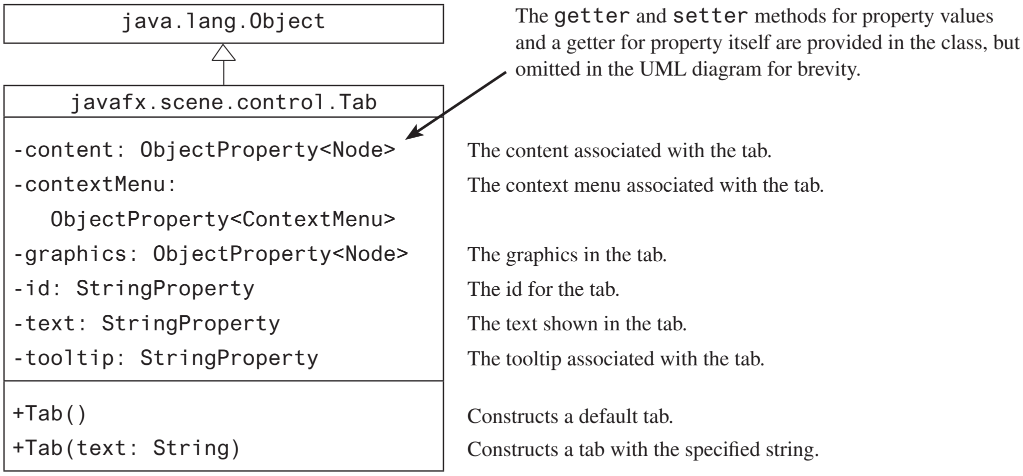
The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The position of the tab in the tab pane. Possible values are: Side.TOP, Side.BOTTOM, Side.LEFT, and Side.RIGHT (default: Side.TOP).

Creates a default Tab Pane.

Returns a list of tabs in this Tab Pane.

FIGURE 31.28 TabPane displays and manages the tabs.



The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The content associated with the tab.

The context menu associated with the tab.

The graphics in the tab.

The id for the tab.

The text shown in the tab.

The tooltip associated with the tab.

Constructs a default tab.

Constructs a tab with the specified string.

Figure 31.29 Tab contains a node.

Listing 31.12 gives an example that uses a tab pane with four tabs to display four types of figures: line, rectangle, rounded rectangle, and oval. You can select a figure to display by clicking the corresponding tab, as shown in Figure 31.27.

Listing 31.12 TabPaneDemo.java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Tab;
import javafx.scene.control.TabPane;
import javafx.scene.layout.StackPane;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Ellipse;
import javafx.scene.shape.Line;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class TabPaneDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    TabPane tabPane = new TabPane();
    Tab tab1 = new Tab("Line");
    StackPane pane1 = new StackPane();
    pane1.getChildren().add(new Line(10, 10, 80, 80));
    tab1.setContent(pane1);
    Tab tab2 = new Tab("Rectangle");
    tab2.setContent(new Rectangle(10, 10, 200, 200));
    Tab tab3 = new Tab("Circle");
    tab3.setContent(new Circle(50, 50, 20));
    Tab tab4 = new Tab("Ellipse");
    tab4.setContent(new Ellipse(10, 10, 100, 80));
    tabPane.getTabs().addAll(tab1, tab2, tab3, tab4);

    Scene scene = new Scene(tabPane, 300, 250);
    primaryStage.setTitle("DisplayFigure"); // Set the window title
    primaryStage.setScene(scene); // Place the scene in the window
    primaryStage.show(); // Display the window
  }
}
```

The program creates a tab pane (line 15) and four tabs (lines 16, 20, 22, and 24). A stack pane is created to hold a line (line 18) and placed into tab1 (line 19). A rectangle, circle, and oval are created and placed into tab2, tab3, and tab4. Note the line is centered in tab1 because it is placed in a stack pane. The other shapes are directly placed into the tab. They are displayed at the upper left corner of the tab.

By default, the tabs are placed at the top of the tab pane. You can use the setSide method to change its location.

Check Point

31.9.1 How do you create a tab pane? How do you create a tab? How do you add a tab to a tab pane?

31.9.2 How do you place the tabs on the left of the tab pane?

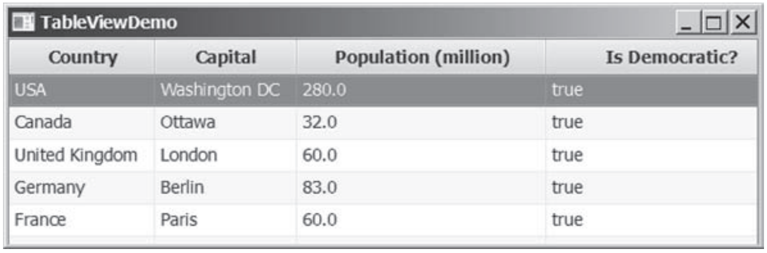
31.9.3 Can a tab have a text as well as an image? Write the code to set an image for tab1 in Listing 31.12.

31.10 TableView

Key Point

You can display tables using the TableView class.

TableView is a control that displays data in rows and columns in a two-dimensional grid, as shown in Figure 31.30.



Country	Capital	Population (million)	Is Democratic?
USA	Washington DC	280.0	true
Canada	Ottawa	32.0	true
United Kingdom	London	60.0	true
Germany	Berlin	83.0	true
France	Paris	60.0	true

FIGURE 31.30 TableView displays data in a table.

TableView, TableColumn, and TableCell are used to display and manipulate a table. TableView displays a table. TableColumn defines the columns in a table. TableCell represents a cell in the table. Creating a TableView is a multistep process. First, you need to create an instance of TableView and associate data with the TableView. Second, you need to create columns using the TableColumn class and set a column cell value factory to specify how to populate all cells within a single TableColumn.

Listing 31.13 gives a simple example to demonstrate using TableView and TableColumn. A sample run of the program is shown in Figure 31.31.

LISTING 31.13 TableViewDemo.java

```
import javafx.application.Application;
import javafx.beans.property.SimpleBooleanProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;
```

```
public class TableViewDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    TableView<Country> tableView = new TableView<>();
    ObservableList<Country> data =
      FXCollections.observableArrayList(
        new Country("USA", "Washington DC", 280, true),
        new Country("Canada", "Ottawa", 32, true),
        new Country("United Kingdom", "London", 60, true),
        new Country("Germany", "Berlin", 83, true),
        new Country("France", "Paris", 60, true));
    tableView.setItems(data);

    TableColumn countryColumn = new TableColumn("Country");
    countryColumn.setMinWidth(100);
    countryColumn.setCellValueFactory(
      new PropertyValueFactory<Country, String>("country"));

    TableColumn capitalColumn = new TableColumn("Capital");
    capitalColumn.setMinWidth(100);
    capitalColumn.setCellValueFactory(
      new PropertyValueFactory<Country, String>("capital"));

    TableColumn populationColumn =
      new TableColumn("Population (million)");
    populationColumn.setMinWidth(200);
    populationColumn.setCellValueFactory(
      new PropertyValueFactory<Country, Double>("population"));

    TableColumn democraticColumn =
      new TableColumn("Is Democratic?");
    democraticColumn.setMinWidth(200);
    democraticColumn.setCellValueFactory(
      new PropertyValueFactory<Country, Boolean>("democratic"));

    tableView.getColumns().addAll(countryColumn, capitalColumn,
      populationColumn, democraticColumn);

    Pane pane = new Pane();
    pane.getChildren().add(tableView);
    Scene scene = new Scene(pane, 300, 250);
    primaryStage.setTitle("TableViewDemo"); // Set the window  title
    primaryStage.setScene(scene); // Place the scene in t he window
    primaryStage.show(); // Display the window
  }

  public static class Country {
    private final SimpleStringProperty country;
    private final SimpleStringProperty capital;
    private final SimpleDoubleProperty population;
    private final SimpleBooleanProperty democratic;

    private Country(String country, String capital,
       double population, boolean democratic) {
      this.country = new SimpleStringProperty(country);
      this.capital = new SimpleStringProperty(capital);
      this.population = new SimpleDoubleProperty(population);
      this.democratic = new SimpleBooleanProperty(democratic);
    }
```

```
 73
 74     public String getCountry() {
 75       return country.get();
 76     }
 77
 78     public void setCountry(String country) {
 79       this.country.set(country);
 80     }
 81
 82     public String getCapital() {
 83       return capital.get();
 84     }
 85
 86     public void setCapital(String capital) {
 87       this.capital.set(capital);
 88     }
 89
 90     public double getPopulation() {
 91       return population.get();
 92     }
 93
 94     public void setPopulation(double population) {
 95       this.population.set(population);
 96     }
 97
 98     public boolean isDemocratic() {
 99       return democratic.get();
100     }
101
102     public void setDemocratic(boolean democratic) {
103       this.democratic.set(democratic);
104     }
105   }
106 }
```

The program creates a TableView (line 17). The TableView class is a generic class whose concrete type is Country. Therefore, this TableView is for displaying Country. The table data is an ObservableList<Country>. The program creates the list (lines 18–24) and associates the list with the TableView (line 25).

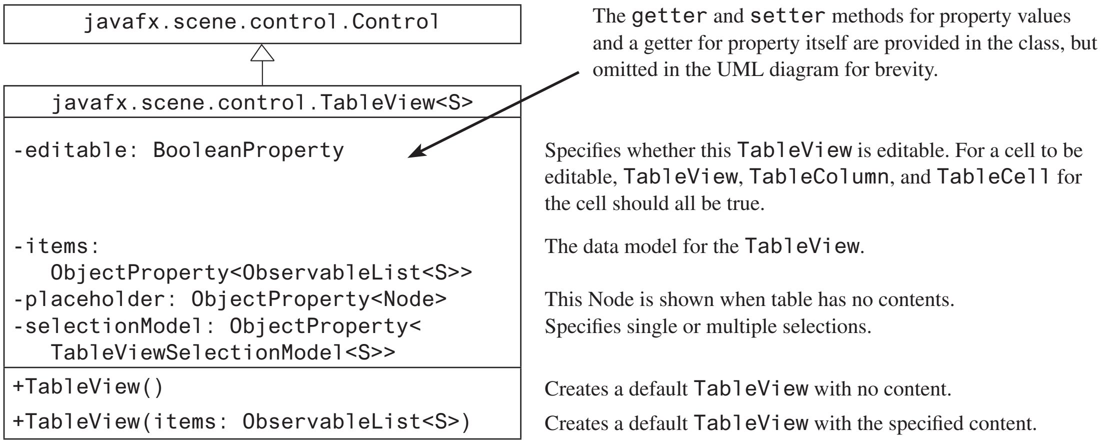
The program creates a TableColumn for each column in the table (lines 27–47). A PropertyValueFactory object is created and set for each column (line 30). This object is used to populate the data in the column. The PropertyValueFactory<S, T> class is a generic class. S is for the class displayed in the TableView and T is the class for the values in the column. The PropertyValueFactory object associates a property in class S with a column.

When you create a table in a JavaFX application, it is a best practice to define the data model in a class. The Country class defines the data for TableView. Each property in the class defines a column in the table. This property should be defined as binding property with the getter and setter methods for the value.

The program adds the columns into the TableView (lines 49 and 50), adds the TableView in a pane (line 53), and places the pane in the scene (line 54). Note line 31 can be simplified using the following code:

```
new PropertyValueFactory<>("country");
```

From this example, you see how to display data in a table using the TableView and TableColumn classes. The frequently used properties and methods for the TableView and TableColumn classes are given in Figures 31.32 and 31.33.



```
javafx.scene.control.Control
            △
javafx.scene.control.TableView<S>
-editable: BooleanProperty
-items:
   ObjectProperty<ObservableList<S>>
-placeholder: ObjectProperty<Node>
-selectionModel: ObjectProperty<
   TableViewSelectionModel<S>>
+TableView()
+TableView(items: ObservableList<S>)
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

Specifies whether this TableView is editable. For a cell to be editable, TableView, TableColumn, and TableCell for the cell should all be true.

The data model for the TableView.

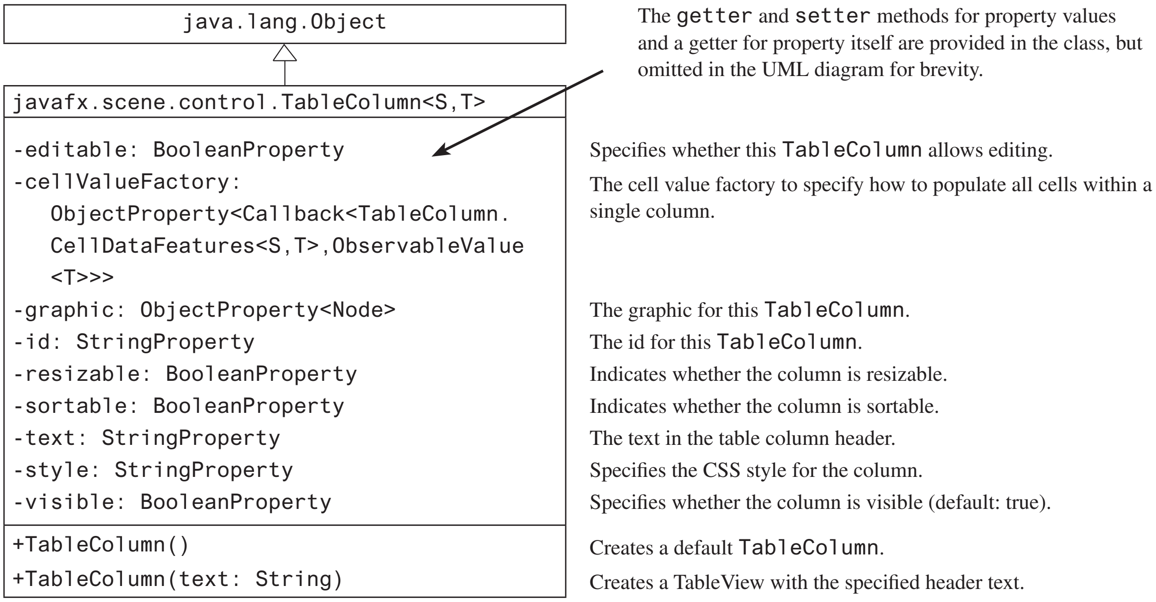
This Node is shown when table has no contents.

Specifies single or multiple selections.

Creates a default TableView with no content.

Creates a default TableView with the specified content.

FIGURE 31.31 TableView displays a table.



```
java.lang.Object
        △
javafx.scene.control.TableColumn<S,T>
-editable: BooleanProperty
-cellValueFactory:
   ObjectProperty<Callback<TableColumn.
   CellDataFeatures<S,T>,ObservableValue
   <T>>>
-graphic: ObjectProperty<Node>
-id: StringProperty
-resizable: BooleanProperty
-sortable: BooleanProperty
-text: StringProperty
-style: StringProperty
-visible: BooleanProperty
+TableColumn()
+TableColumn(text: String)
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

Specifies whether this TableColumn allows editing.

The cell value factory to specify how to populate all cells within a single column.

The graphic for this TableColumn.

The id for this TableColumn.

Indicates whether the column is resizable.

Indicates whether the column is sortable.

The text in the table column header.

Specifies the CSS style for the column.

Specifies whether the column is visible (default: true).

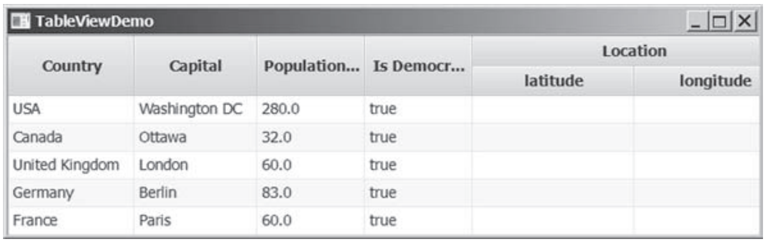
Creates a default TableColumn.

Creates a TableView with the specified header text.

FIGURE 31.32 TableColumn defines a column in the TableView.

You can create nested columns. For example, the following code creates two subcolumns under Location, as shown in Figures 31.33.

```
TableColumn locationColumn = new TableColumn("Location");
locationColumn.getColumns().addAll(new TableColumn("latitude"),
  new TableColumn("longitude"));
```



Country	Capital	Population...	Is Democr...	Location	
				latitude	longitude
USA	Washington DC	280.0	true		
Canada	Ottawa	32.0	true		
United Kingdom	London	60.0	true		
Germany	Berlin	83.0	true		
France	Paris	60.0	true		

FIGURE 31.33 You can add subcolumns in a column.

The TableView data model is an observable list. When data is changed, the change is automatically shown in the table. Listing 31.14 gives an example that lets the user add new rows to the table.

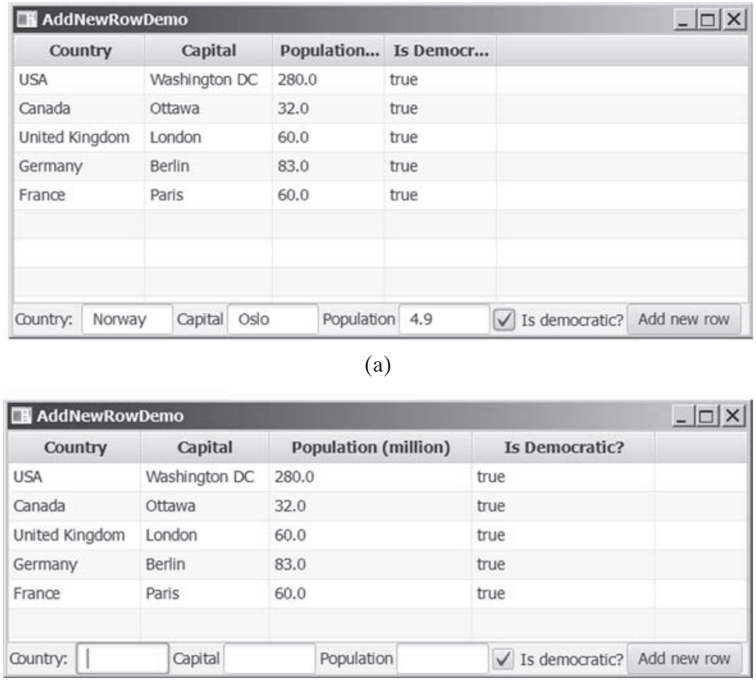
Listing 31.14 AddNewRowDemo.java

```java
import javafx.application.Application;
import javafx.beans.property.SimpleBooleanProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.CheckBox;
import javafx.scene.control.Label;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.TextField;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;

public class AddNewRowDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    TableView<Country> tableView = new TableView<>();
    ObservableList<Country> data =
      FXCollections.observableArrayList(
        new Country("USA", "Washington DC", 280, true),
        new Country("Canada", "Ottawa", 32, true),
        new Country("United Kingdom", "London", 60, true),
        new Country("Germany", "Berlin", 83, true),
        new Country("France", "Paris", 60, true));
    tableView.setItems(data);

    TableColumn countryColumn = new TableColumn("Country");
    countryColumn.setMinWidth(100);
    countryColumn.setCellValueFactory(
      new PropertyValueFactory<Country, String>("country"));

    TableColumn capitalColumn = new TableColumn("Capital");
    capitalColumn.setMinWidth(100);
    capitalColumn.setCellValueFactory(
      new PropertyValueFactory<Country, String>("capital"));

    TableColumn populationColumn =
      new TableColumn("Population (million)");
    populationColumn.setMinWidth(100);
    populationColumn.setCellValueFactory(
      new PropertyValueFactory<Country, Double>("population"));

    TableColumn democraticColumn =
      new TableColumn("Is Democratic?");
    democraticColumn.setMinWidth(100);
    democraticColumn.setCellValueFactory(
      new PropertyValueFactory<Country, Boolean>("democratic"));

    tableView.getColumns().addAll(countryColumn, capitalColumn,
      populationColumn, democraticColumn);

    FlowPane flowPane = new FlowPane(3, 3);
    TextField tfCountry = new TextField();
    TextField tfCapital = new TextField();
    TextField tfPopulation = new TextField();
    CheckBox chkDemocratic = new CheckBox("Is democratic?");
    Button btAddRow = new Button("Add new row");
    tfCountry.setPrefColumnCount(5);
    tfCapital.setPrefColumnCount(5);
    tfPopulation.setPrefColumnCount(5);
    flowPane.getChildren().addAll(new Label("Country: "),
      tfCountry, new Label("Capital"), tfCapital,
      new Label("Population"), tfPopulation, chkDemocratic,
      btAddRow);

    btAddRow.setOnAction(e -> {
      data.add(new Country(tfCountry.getText(), tfCapital.getText(),
        Double.parseDouble(tfPopulation.getText()),
        chkDemocratic.isSelected()));
      tfCountry.clear();
      tfCapital.clear();
      tfPopulation.clear();
    });

    BorderPane pane = new BorderPane();
    pane.setCenter(tableView);
    pane.setBottom(flowPane);

    Scene scene = new Scene(pane, 500, 250);
    primaryStage.setTitle("AddNewRowDemo"); // Set the window title
    primaryStage.setScene(scene); // Place the scene in the window
    primaryStage.show(); // Display the window
  }

  public static class Country {
    private final SimpleStringProperty country;
    private final SimpleStringProperty capital;
    private final SimpleDoubleProperty population;
    private final SimpleBooleanProperty democratic;

    private Country(String country, String capital,
      double population, boolean democratic) {
    this.country = new SimpleStringProperty(country);
    this.capital = new SimpleStringProperty(capital);
    this.population = new SimpleDoubleProperty(population);
    this.democratic = new SimpleBooleanProperty(democratic);
  }

  public String getCountry() {
    return country.get();
  }

  public void setCountry(String country) {
    this.country.set(country);
  }

  public String getCapital() {
    return capital.get();
  }
```

```
115
116       public void setCapital(String capital) {
117         this.capital.set(capital);
118       }
119
120       public double getPopulation() {
121         return population.get();
122       }
123
124       public void setPopulation(double population) {
125         this.population.set(population);
126       }
127
128       public boolean isDemocratic() {
129         return democratic.get();
130       }
131
132       public void setDemocratic(boolean democratic) {
133         this.democratic.set(democratic);
134       }
135     }
136   }
```

The program is the same in Listing 31.13 except that a new code is added to let the user enter a new row (lines 57–82). The user enters the new row from the text fields and a check box and presses the *Add New Row* button to add a new row to the data. Since data is an observable list, the change in data is automatically updated in the table.

As shown in Figure 31.34a, a new country information is entered in the text fields. After clicking the *Add New Row button*, the new country is displayed in the table view.



Country	Capital	Population...	Is Democr...
USA	Washington DC	280.0	true
Canada	Ottawa	32.0	true
United Kingdom	London	60.0	true
Germany	Berlin	83.0	true
France	Paris	60.0	true

Country: Norway Capital Oslo Population 4.9 ☑ Is democratic? Add new row

(a)

Country	Capital	Population (million)	Is Democratic?
USA	Washington DC	280.0	true
Canada	Ottawa	32.0	true
United Kingdom	London	60.0	true
Germany	Berlin	83.0	true
France	Paris	60.0	true

Country: Capital Population ☑ Is democratic? Add new row

(b)

FIGURE 31.34 Change in the table data model is automatically displayed in the table view.

TableView not only displays data, but also allows data to be edited. To enable data editing in the table, write the code as follows:

1. Set the TableView's editable to true.
2. Set the column's cell factory to a text field table cell.
3. Implement the column's setOnEditCommit method to assign the edited value to the data model.

   Here is the example of enabling editing for the countryColumn.

```
tableView.setEditable(true);
countryColumn.setCellFactory(TextFieldTableCell.forTableColumn());
countryColumn.setOnEditCommit(
  new EventHandler<CellEditEvent<Country, String>>() {
    @Override
    public void handle(CellEditEvent<Country, String> t) {
      t.getTableView().getItems().get(
        t.getTablePosition().getRow())
      .setCountry(t.getNewValue());
    }
  }
);
```

Check Point

31.10.1 How do you create a table view? How do you create a table column? How do you add a table column to a table view?

31.10.2 What is the data type for a TableView's data model? How do you associate a data model with a TableView?

31.10.3 How do you set a cell value factory for a TableColumn?

31.10.4 How do you set an image in a table column header?

31.11 Developing JavaFX Programs Using FXML

Key Point

You can create JavaFX user interfaces using FXML scripts.

There are two ways to develop JavaFX applications. One way is to write everything in Java source code as you have done so far. The other way is to use FXML. FXML is an XML-based script language for describing the user interface. Using FXML enables you to separate user interface from the logic of the Java code. JavaFX Scene Builder is a visual design tool for creating the user interface without manually writing the FXML script. You drag and drop the UI components to the content pane and set properties for the components in the Inspector. The Scene Builder automatically generates the FXML scripts for the user interface. This section demonstrates how to use the Scene Builder to create JavaFX applications.

NOTE
It is important that you first learn how to write the JavaFX code without using FXML to grasp the fundamentals of JavaFX programming before learning FXML. Once you understand the basics of JavaFX, it is easy to create JavaFX programs using FXML. For this reason, FXML is introduced after you have learned the basics of JavaFX programming.

31.11.1 Installing JavaFX Scene Builder

You can use the JavaFX Scene Builder standalone or with an IDE such as NetBeans or Eclipse. This section demonstrates using the JavaFX Scene Builder with NetBeans. You can download the latest version of Scene Builder from http://gluonhq.com/open-source/scene-builder/.

31.11.2 Creating a JavaFX FXML Project

To use JavaFX FXML, you need to create a JavaFX FXML in NetBeans. Here are the steps of creating a JavaFX FXML project:

1. Choose *File*, *New Project* to display the New Project dialog box, as shown in Figure 31.35.
2. Choose *JavaFX* in the Categories and *JavaFX FXML Application* in the Projects. Click *Next* to display the New JavaFX Application dialog box, as shown in Figure 31.36.
3. Enter Calculator as the project name and click *Finish* to create the project. You will see the project created as shown in Figure 31.37.

Three files, `Calculator.java`, `FXMLDocument.fxml`, and `FXMLDocumentController.java`, are created in the project. Their source codes are shown in Listings 31.15, 31.16, and 31.17. From the perspective of the MVC architecture, these three files correspond to model, view, and controller. You can define data model in the `Calculator.java` class. The .fxml file describes the user interface. The controller file defines the actions for processing the events for the user interface.

Listing 31.15 Calculator.java

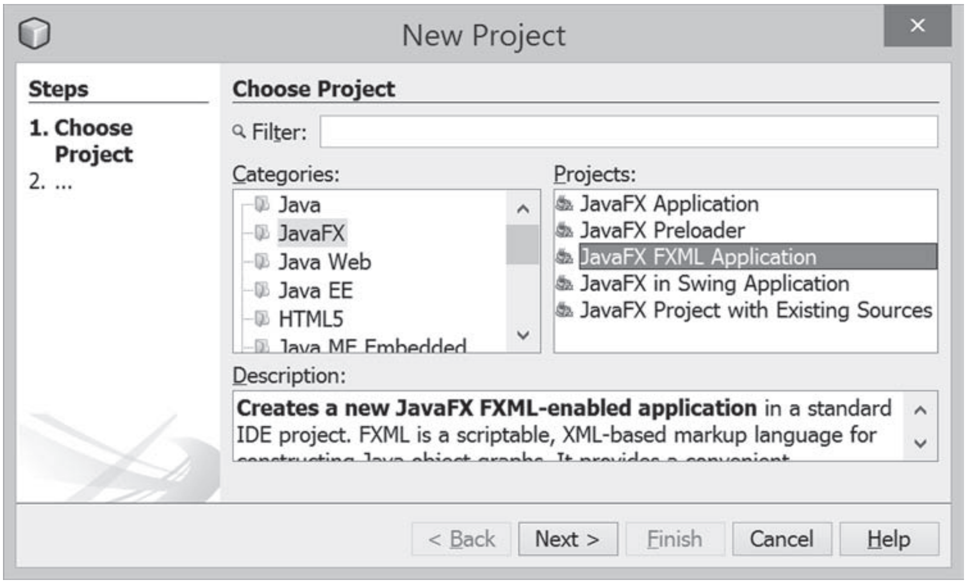
```java
package calculator;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Calculator extends Application {
  @Override
  public void start(Stage stage) throws Exception {
    Parent root =
      FXMLLoader.load(getClass().getResource("FXMLDocument.fxml"));
    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.show();
  }

  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    launch(args);
  }
}
```

Listing 31.16 FXMLDocument.fxml

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

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

<AnchorPane id="AnchorPane" prefHeight="200" prefWidth="320"
            xmlns:fx="http://javafx.com/fxml/1"
            fx:controller="calculator.FXMLDocumentController">
    <children>
        <Button layoutX="126" layoutY="90" text="Click Me!"
                onAction="#handleButtonAction" fx:id="button" />
        <Label layoutX="126" layoutY="120" minHeight="16" minWidth="69"
               fx:id="label" />
     </children>
</AnchorPane>
```

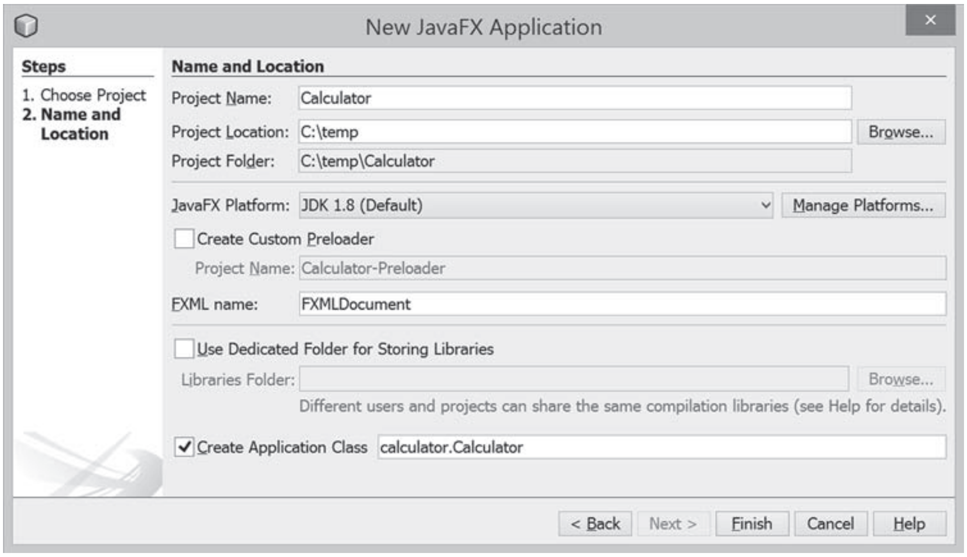
LISTING 31.17 FXMLDocumentController.java

```
package calculator;

import java.net.URL;
import java.util.ResourceBundle;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Label;

public class FXMLDocumentController implements Initializable {
  @FXML
  private Label label;

  @FXML
  private void handleButtonAction(ActionEvent event) {
    System.out.println("You clicked me!");
    label.setText("Hello World!");
  }

  @Override
  public void initialize(URL url, ResourceBundle rb) {
    // TODO
  }
}
```



Creates a new JavaFX FXML-enabled application in a standard IDE project. FXML is a scriptable, XML-based markup language for

FIGURE 31.35 You can choose JavaFX in the Categories and JavaFX FXML Application in the Project to create a FXML project.


New JavaFX Application

Steps
1. Choose Project
2. Name and Location

Name and Location

Project Name: Calculator

Project Location: C:\temp

Project Folder: C:\temp\Calculator

JavaFX Platform: JDK 1.8 (Default)

□ Create Custom Preloader

Project Name: Calculator-Preloader

FXML name: FXMLDocument

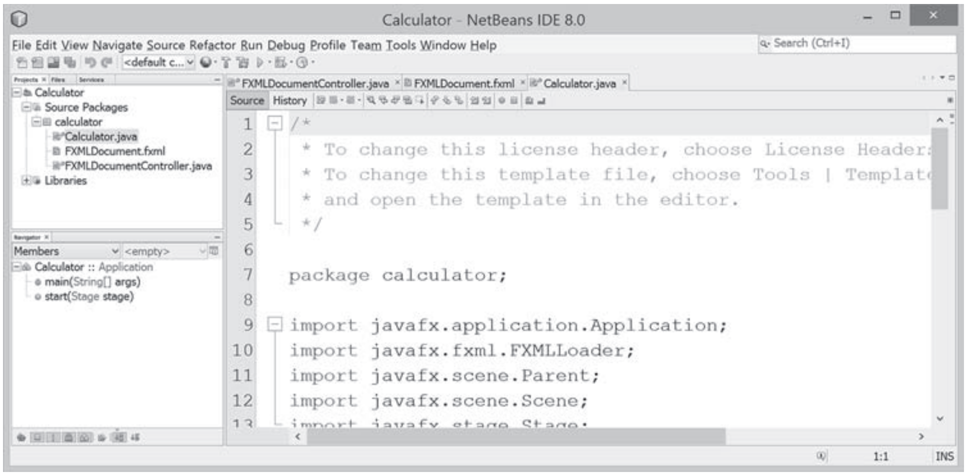
□ Use Dedicated Folder for Storing Libraries

Libraries Folder:

Different users and projects can share the same compilation libraries (see Help for details).

☑ Create Application Class calculator.Calculator

FIGURE 31.36 You can enter project information in the New JavaFX Application dialog.


Calculator - NetBeans IDE 8.0

```
/*
 * To change this license header, choose License Header
 * To change this template file, choose Tools | Template
 * and open the template in the editor.
 */

package calculator;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;
```

FIGURE 31.37 A FXML project is created.

31.11.3 Creating User Interfaces

We now turn our attention to developing a simple calculator as shown in Figure 31.38. The Calculator program enables the user to enter numbers and perform addition, subtraction, multiplication, and division.

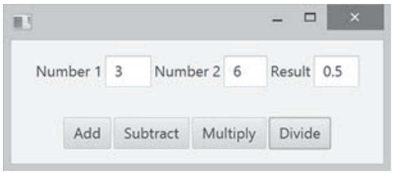

Number 1 3 Number 2 6 Result 0.5

FIGURE 31.38 The application performs arithmetic operations.

When you create a JavaFX FXML project, NetBeans creates a default .fxml file that contains the contents for a simple sample user interface. To view the user interface, double-click the .fxml file to open the Scene Builder, as shown in Figure 31.39. Note NetBeans can automatically detect the Scene Builder after it is installed on your machine.

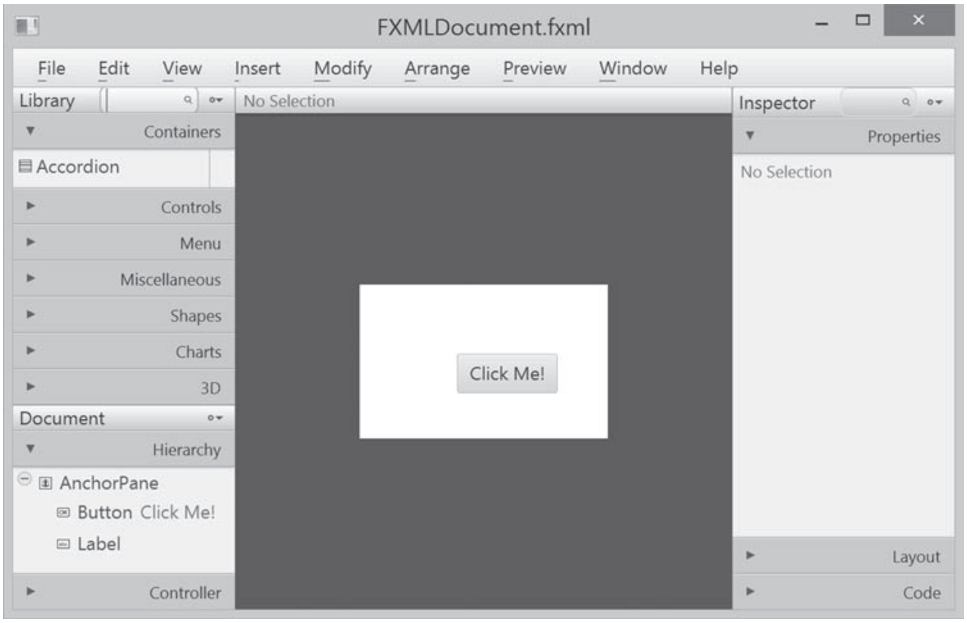


FIGURE 31.39 Double-click the .fxml file to open the Scene Builder.

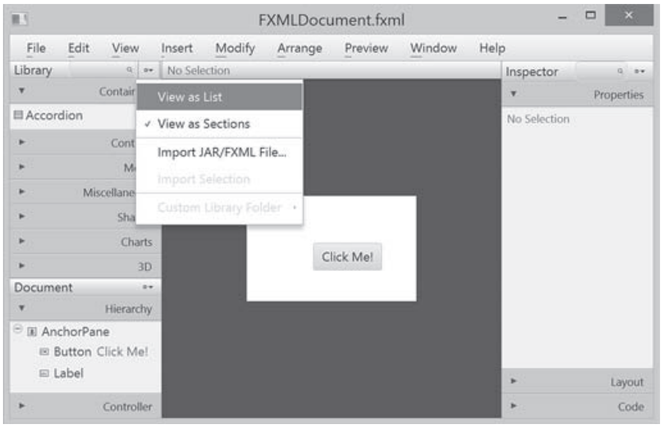


FIGURE 31.40 You can open the Library pane by clicking the Library icon and choose View as List.

To start a new user interface, delete the default user interface in the .fxml file from the content pane, as shown in Figure 31.41. Here are the steps to create a new user interface:

1. (Optional) On some systems, the components in the Library pane are not visible by sections. Click the Library icon to open the context menu as shown in Figure 31.40 and choose *View as List*.
2. Drag a `BorderPane` into the user interface and drag an `HBox` to the center of the `BorderPane` and another `HBox` to the bottom of the `BorderPane`. Set the alignment of both `HBox` to `CENTER` as shown in Figure 31.42. Set the `Spacing` property in the Layout section of the Inspector to 5. When you select a component in the visual layout, the properties of the component are displayed in the Inspector pane, where you can set the properties.

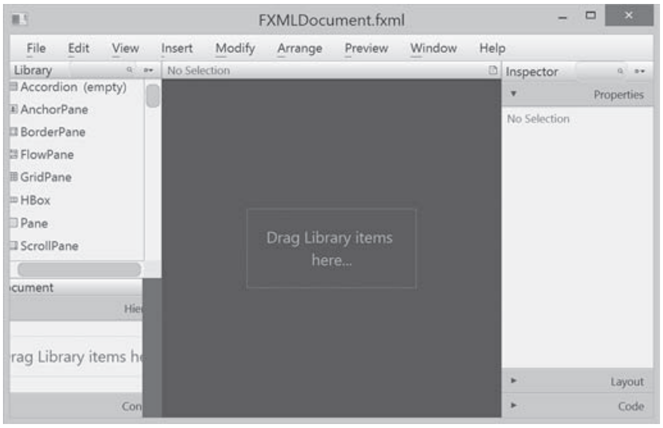


FIGURE 31.41 The UI is empty after deleting the default button in the pane.

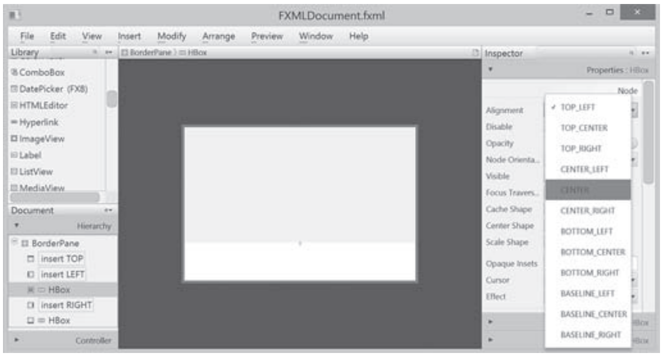


FIGURE 31.42 A BorderPane is dropped to the UI and an HBox is placed at the bottom of the BorderPane.

3. Drag and drop a **Label**, a **TextField**, a **Label**, a **TextField**, a **Label**, and a **TextField** and change the label's text to Number 1, Number 2, and Result, as shown in Figure 31.43. Set the Pref Column Count property for each text field to 2 in the Layout section of the Inspector. In the Code section of the Inspector, set the id for the text fields to **tfNumber1**, **tfNumber2**, and **tfResult**, as shown in Figure 31.44. These ids are useful to reference the text fields and obtain their values in the controller.

4. Drag and drop four **Buttons** to the second **HBox** and set their text property to Add, Subtract, Multiply, and Divide, as shown in Figure 31.45.

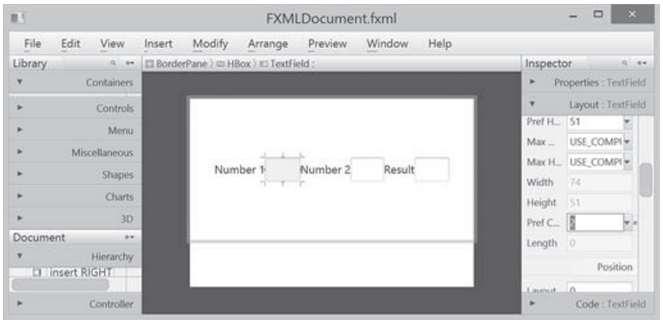
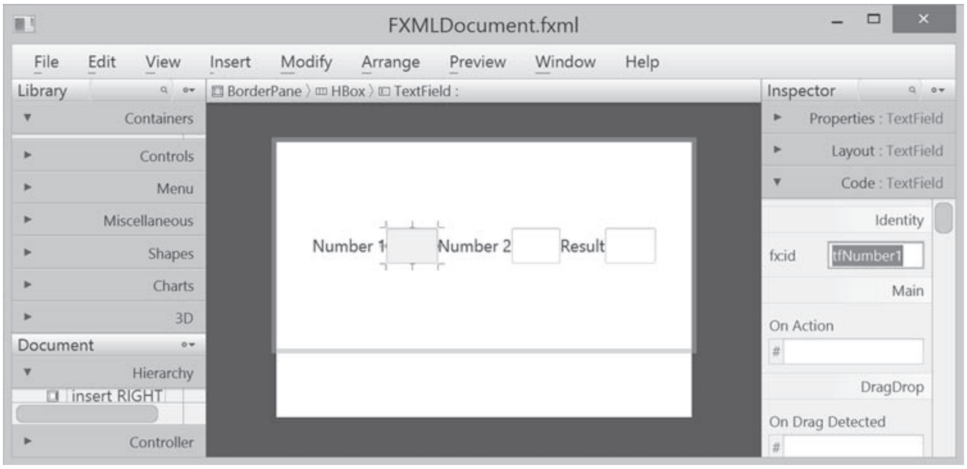
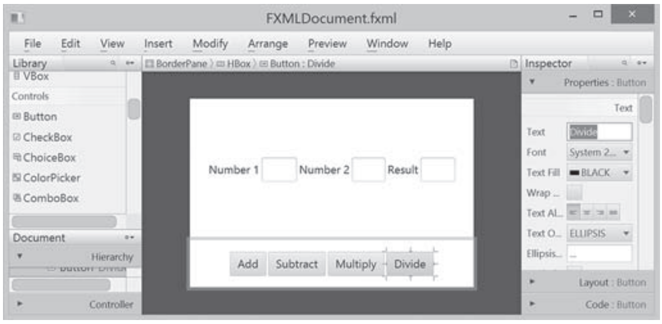


FIGURE 31.43 The labels and text fields are dropped to the UI.



fx:id tfNumber1

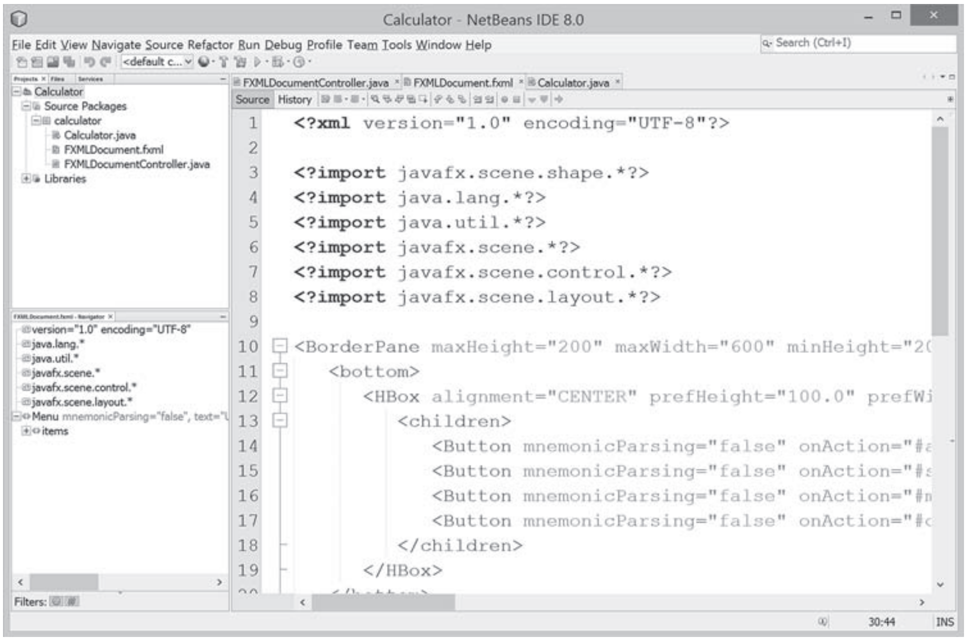
FIGURE 31.44 Set the appropriate id for the text fields.



Text Divide
Font System 2...
Text Fill BLACK
Text O... ELLIPSIS

FIGURE 31.45 The buttons are dropped to the HBox.

After you create and make changes to the user interface in the content pane, you need to save the changes by choosing *File*, *Save* from the menu bar in the Scene Builder. The .fxml file is updated and synchronized with the changes in the content pane. You can view the contents in the .fxml file from NetBeans, as shown in Figure 31.46.



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

<?import javafx.scene.shape.*?>
<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<BorderPane maxHeight="200" maxWidth="600" minHeight="2(
    <bottom>
        <HBox alignment="CENTER" prefHeight="100.0" prefWi
            <children>
                <Button mnemonicParsing="false" onAction="#a
                <Button mnemonicParsing="false" onAction="#s
                <Button mnemonicParsing="false" onAction="#m
                <Button mnemonicParsing="false" onAction="#c
            </children>
        </HBox>
```

FIGURE 31.46 You can view the contents of the FXML file.

31.11.4 Handling Events in the Controller

The .fxml file describes the user interface. You write the code to implement the logic in the controller file, as shown in Listing 31.18.

Listing 31.18 FXMLDocumentController.java

```
 1  package calculator;
 2
 3  import javafx.event.ActionEvent;
 4  import javafx.fxml.FXML;
 5  import javafx.scene.control.TextField;
 6
 7  public class FXMLDocumentController {
 8    @FXML
 9    private TextField tfNumber1, tfNumber2, tfResult;
10
11    @FXML
12    private void addButtonAction(ActionEvent event) {
13      tfResult.setText(getResult('+') + "");
14    }
15
16    @FXML
17    private void subtractButtonAction(ActionEvent event) {
18      tfResult.setText(getResult('-') + "");
19    }
20
21    @FXML
22    private void multiplyButtonAction(ActionEvent event) {
23      tfResult.setText(getResult('*') + "");
24    }
25
26    @FXML
27    private void divideButtonAction(ActionEvent event) {
28      tfResult.setText(getResult('/') + "");
29    }
30
31    private double getResult(char op) {
32      double number1 = Double.parseDouble(tfNumber1.getText());
33      double number2 = Double.parseDouble(tfNumber2.getText());
34      switch (op) {
35        case '+':  return number1 + number2;
36        case '-':  return number1 - number2;
37        case '*':  return number1 * number2;
38        case '/':  return number1 / number2;
39      }
40      return Double.NaN;
41    }
42  }
```

The controller class declares three `TextFields`, `tfNumber1`, `tfNumber2`, and `tfResult` (line 9). The @FXML annotation denotes that these data fields are linked to the text fields in the user interface. Recall in the user interface, we set the id for the three text fields as `tfNumber1`, `tfNumber2`, and `tfResult`.

The codes for handling the events from the buttons are defined in the methods `addButtonAction`, `subtractButtonAction`, `multiplyButtonAction`, and `divideButtonAction` (lines 11–29). The @FXML annotation is used to denote that these methods will be tied to the button actions in the view.

Through the @FXML annotation, the data fields and methods in the controller are linked to the components and actions defined in the .fxml file.

31.11.5 Linking View with Controller

You can now link the actions from the components in the view with the processing methods in the controller. Here are the steps to accomplish it:

1. Add the following attribute in the <BorderPane> tag for using a controller with the view.

   ```
   fx:controller="calculator.FXMLDocumentController"
   ```

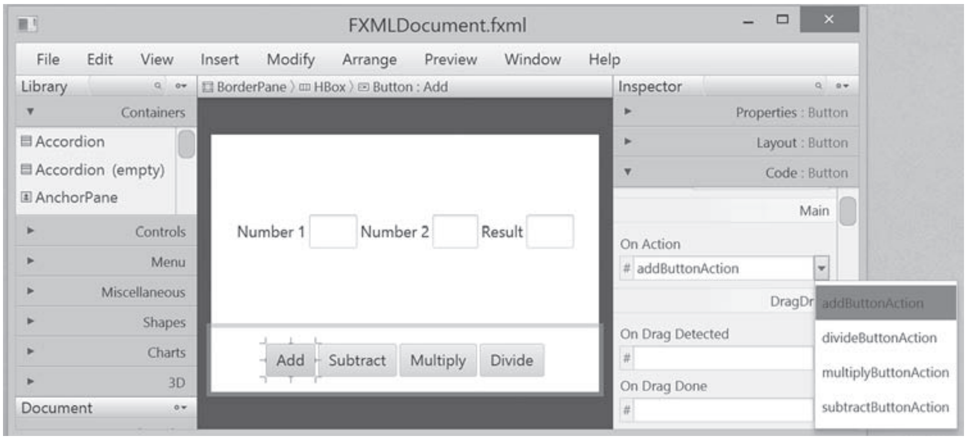
2. Double-click the .fxml file in the project to display the visual layout window. In the Inspector for the *Add* button, choose **addButtonAction** from a list of action processing methods, as shown in Figure 31.47. The complete code for the .fxml file is shown in Listing 31.19.

LISTING 31.19 FXMLDocument.fxml

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

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


<BorderPane maxHeight="200" maxWidth="600" minHeight="200"
            minWidth="600" prefHeight="400.0" prefWidth="600.0"
            xmlns="http://javafx.com/javafx/8"
            xmlns:fx="http://javafx.com/fxml/1"
            fx:controller="calculator.FXMLDocumentController">
   <bottom>
      <HBox alignment="CENTER" prefHeight="100.0" prefWidth="200.0"
           spacing="5.0" BorderPane.alignment="CENTER">
        <children>
           <Button mnemonicParsing="false"
              onAction="#addButtonAction" text="Add" />
           <Button mnemonicParsing="false"
              onAction="#subtractButtonAction" text="Subtract" />
           <Button mnemonicParsing="false"
              onAction="#multiplyButtonAction" text="Multiply" />
           <Button mnemonicParsing="false"
              onAction="#divideButtonAction" text="Divide" />
        </children>
     </HBox>
   </bottom>
   <center>
      <HBox alignment="CENTER" prefHeight="232.0" prefWidth="572.0"
           spacing="5.0" BorderPane.alignment="CENTER">
        <children>
           <Label text="Number 1" />
           <TextField fx:id="tfNumber1" prefColumnCount="2"
              prefHeight="51.0" prefWidth="74.0" />
           <Label text="Number 2" />
           <TextField fx:id="tfNumber2" prefColumnCount="2"
              prefHeight="51.0" prefWidth="70.0" />
           <Label text="Result" />
           <TextField fx:id="tfResult" prefColumnCount="2" />
        </children>
      </HBox>
   </center>
</BorderPane>
```

31.11.6 Running the Project

The code in the model is automatically generated as shown in Listing 31.15. This is the main program that loads the FXML to create the user interface in a Parent object (lines 12 and 13). The parent object is then added to the scene (line 14). The scene is set to the stage (line 15). The stage is displayed in line 16.



On Action
# addButtonAction

Figure 31.47 Choosing addButtonAction to generate the code for handling action for the Add button.

Chapter Summary

1. JavaFX provides the cascading style sheets based on CSS. You can use the `getStylesheets` method to load a style sheet and use the `setStyle`, `setStyleClass`, and `setId` methods to set JavaFX CSS for nodes.
2. JavaFX provides the `QuadCurve`, `CubicCurve`, and `Path` classes for creating advanced shapes.
3. JavaFX supports coordinate transformations using translation, rotation, and scaling.
4. You can specify the pattern for a stroke, how the lines are joined in a stroke, the width of a stroke, and the type of a stroke.
5. You can create menus using the `Menu`, `MenuItem`, `CheckMenuItem`, and `RadioMenuItem` classes.
6. You can create context menus using the `ContextMenu` class.
7. The `SplitPane` can be used to display multiple panes horizontally or vertically and allows the user to adjust the sizes of the panes.
8. The `TabPane` can be used to display multiple panes with tabs for selecting panes.
9. You can create and display tables using the `TableView` and `TableColumn` classes.
10. You can create JavaFX user interfaces using FXML. FXML is XML-based script language for describing the user interface. Using FXML enables you to separate user interface from the logic of Java code.
11. JavaFX Scene Builder is a visual tool for creating the user interface without manually writing the FXML scripts.

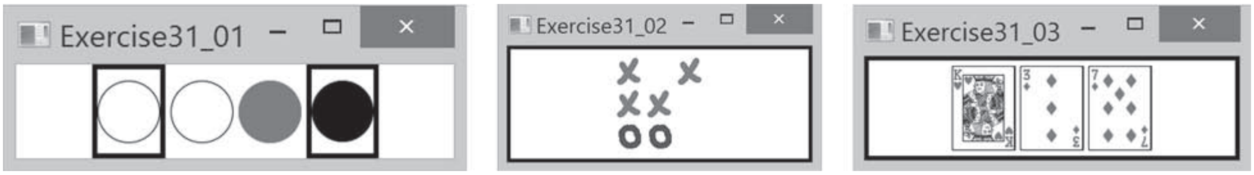
Quiz

Answer the quiz for this chapter online at the book Companion Website.

Programming Exercises

Sections 31.2

31.1 (*Use JavaFX CSS*) Create a CSS style sheet that defines a class for white fill and black stroke color and an id for red stroke and green color. Write a program that displays four circles and uses the style class and id. The sample run of the program is shown in Figure 31.48a.



(a) (b) (c)

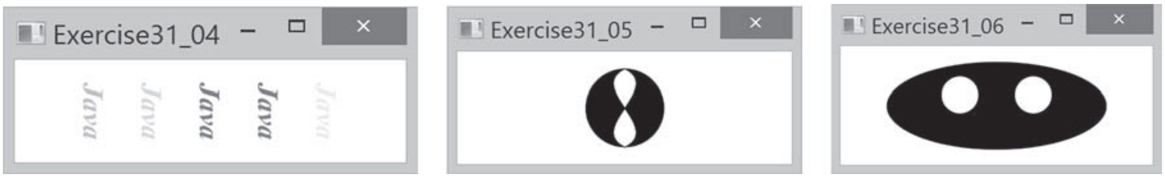
Figure 31.48 (a) The border and the color style for the shapes are defined in a style class. (b) Exercise 31.2 displays a tic-tac-toe board with images using style sheet for border. (c) Three cards are randomly selected.

***31.2** (*Tic-tac-toe board*) Write a program that displays a tic-tac-toe board, as shown in Figure 31.48b. A cell may be X, O, or empty. What to display at each cell is randomly decided. The X and O are images in the files `x.gif` and `o.gif`. Use the style sheet for border.

***31.3** (*Display three cards*) Write a program that displays three cards randomly selected from a deck of 52, as shown in Figure 31.48c. The card image files are named 1.png, 2.png, . . ., 52.png and stored in the `image/card` directory. All the three cards are distinct and selected randomly. Hint: You can select random cards by storing the numbers 1–52 to an array, perform a random shuffle using Section 7.2.6, and use the first three numbers in the array as the file names for the image. Use the style sheet for border.

Sections 31.3

31.4 (*Color and font*) Write a program that displays five texts vertically, as shown in Figure 31.49a. Set a random color and opacity for each text and set the font of each text to Times Roman, bold, italic, and 22 pixels.



(a) (b) (c)

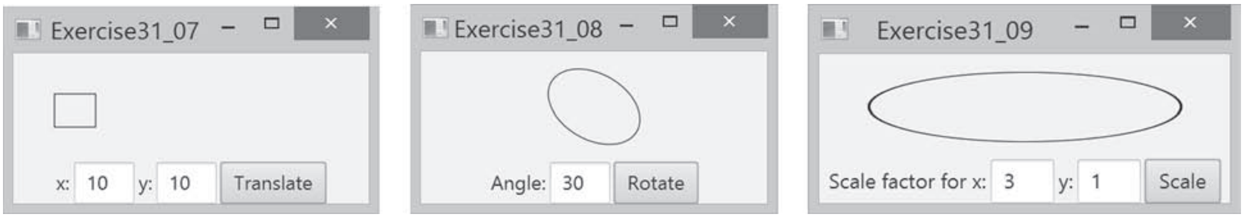
Figure 31.49 (a) Five texts are displayed with a random color and a specified font. (b) A path is displayed inside the circle. (c) Two circles are displayed in an oval.

***31.5** (*Cubic curve*) Write a program that creates two shapes: a circle and a path consisting of two cubic curves, as shown in Figure 31.49b.

***31.6** (*Eyes*) Write a program that displays two eyes in an oval, as shown in Figure 31.49c.

Sections 31.4

***31.7** (*Translation*) Write a program that displays a rectangle with upper-left corner point at (**40**, **40**), width **50**, and height **40**. Enter the values in the text fields *x* and *y* and press the *Translate* button to translate the rectangle to a new location, as shown in Figure 31.50a.



x: 10 y: 10

Angle: 30

Scale factor for x: 3 y: 1

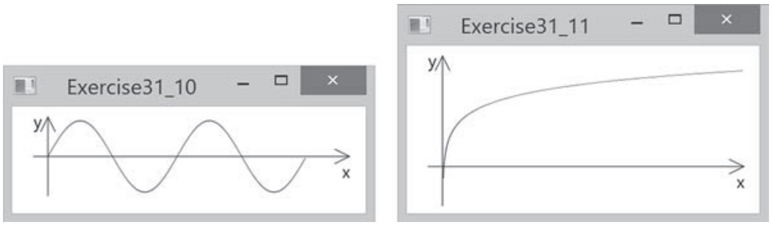
(a) (b) (c)

Figure 31.50 (a) Exercise 31.7 translates coordinates. (b) Exercise 31.8 rotates coordinates. (c) Exercise 31.9 scales coordinates.

***31.8** (*Rotation*) Write a program that displays an ellipse. The ellipse is centered in the pane with width **60** and height **40**. Enter the value in the text field Angle and press the *Rotate* button to rotate the ellipse, as shown in Figure 31.50b.

***31.9** (*Scale graphics*) Write a program that displays an ellipse. The ellipse is centered in the pane with width **60** and height **40**. Enter the scaling factors in the text fields and press the *Scale* button to scale the ellipse, as shown in Figure 31.50c.

***31.10** (*Plot the sine function*) Write a program that plots the sine function, as shown in Figure 31.51a.

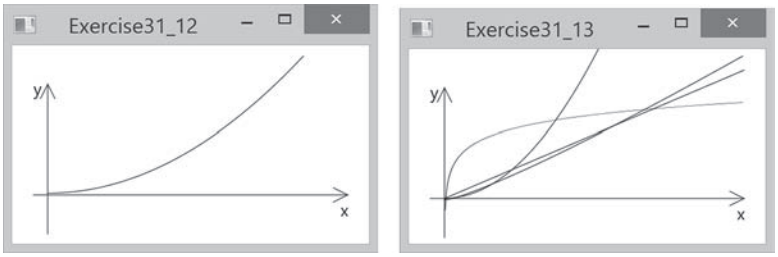


(a) (b)

Figure 31.51 (a) Exercise 31.10 displays a sine function. (b) Exercise 31.11 displays the log function.

***31.11** (*Plot the log function*) Write a program that plots the log function, as shown in Figure 31.51a.

***31.12** (*Plot the n^2 function*) Write a program that plots the n^2 function, as shown in Figure 31.51b 2a.



(a) (b)

Figure 31.52 (a) Exercise 31.13 displays the n^2 function. (b) Exercise 31.13 displays several functions.

***31.13** (*Plot the log, n, nlogn, and n^2 functions*) Write a program that plots the log, *n*, *n*log*n*, and n^2 functions, as shown in Figure 31.52b.

***31.14** (*Scale and rotate graphics*) Write a program that enables the user to scale and rotate the STOP sign, as shown in Figure 31.53. The user can press the UP/DOWN arrow key to increase/decrease the size and press the RIGHT/LEFT arrow key to rotate left or right.

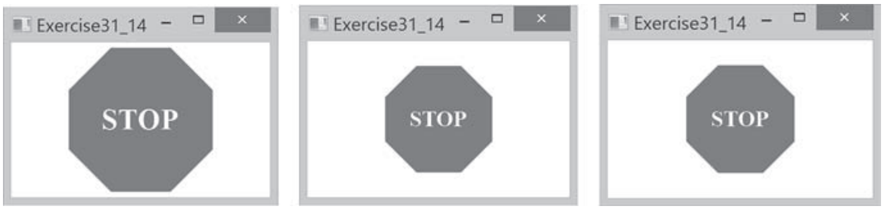
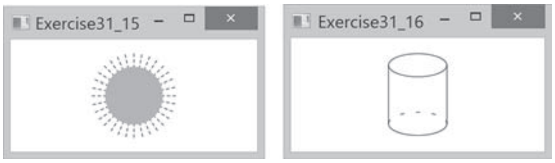


Figure 31.53 The program can rotate and scale the painting.

Sections 31.5

***31.15** (*Sunshine*) Write a program that displays a circle filled with a gradient color to animate a sun and display light rays coming out from the sun using dashed lines, as shown in Figure 31.54a.



(a) (b)

Figure 31.54 (a) Exercise 31.15 displays the sunshine. (b) Exercise 31.16 displays a cylinder.

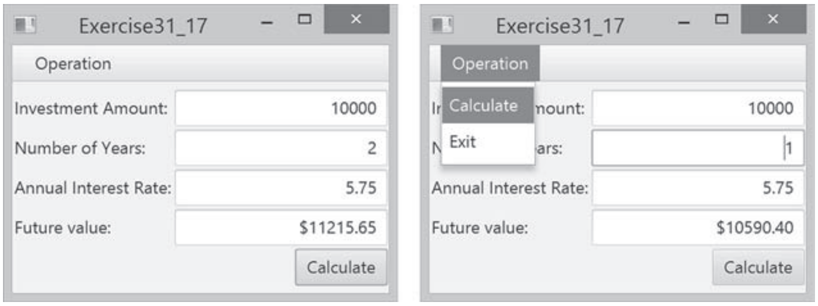
***31.16** (*Display a cylinder*) Write a program that displays a cylinder, as shown in Figure 31.54b. Use dashed strokes to draw the dashed arc.

Sections 31.6

***31.17** (*Create an investment value calculator*) Write a program that calculates the future value of an investment at a given interest rate for a specified number of years. The formula for the calculation is as follows:

$$\texttt{futureValue = investmentAmount} \times \texttt{(1 + monthlyInterestRate)}^{\texttt{years}\times 12}$$

Use text fields for interest rate, investment amount, and years. Display the future amount in a text field when the user clicks the *Calculate* button or chooses Calculate from the Operation menu (see Figure 31.55). Click the Exit menu to exit the program.



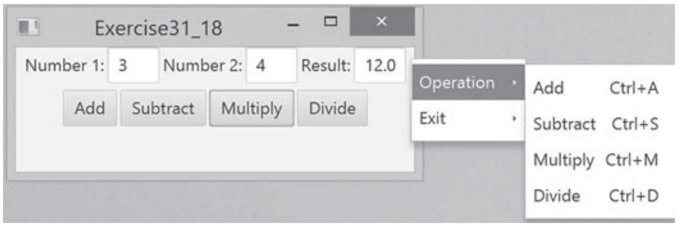
Investment Amount: 10000
Number of Years: 2
Annual Interest Rate: 5.75
Future value: $11215.65

Investment Amount: 10000
Number of Years: 1
Annual Interest Rate: 5.75
Future value: $10590.40

FIGURE 31.55 The user enters the investment amount, years, and interest rate to compute future value.

Sections 31.8

***31.18** (*Use popup menus*) Modify `Listing 31.9`, `MenuDemo.java`, to create a popup menu that contains the menus Operations and Exit, as shown in Figure 31.56. The popup is displayed when you click the right mouse button on the panel that contains the labels and the text fields.



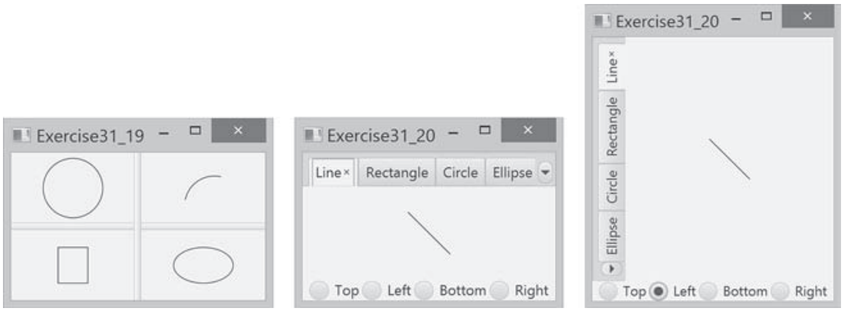
Number 1: 3 Number 2: 4 Result: 12.0

FIGURE 31.56 The popup menu contains the commands to perform arithmetic operations.

***31.19** (*Use* `SplitPane`) Create a program that displays four shapes in split panes, as shown in Figure 31.57a.

Sections 31.9

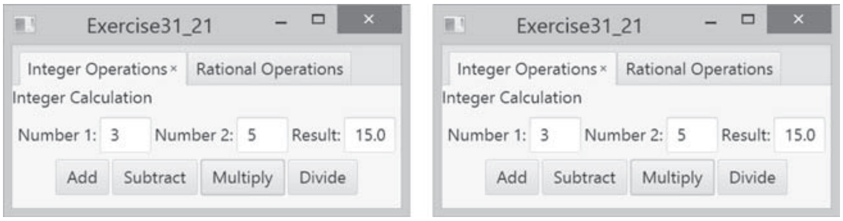
***31.20** (*Use tab panes*) Modify `Listing 31.12`, `TabPaneDemo.java`, to add a pane of radio buttons for specifying the tab placement of the tab pane, as shown in Figure 31.57b and c.



(a) (b) (c)

Figure 31.57 (a) Four shapes are displayed in split panes. (b and c) The radio buttons let you choose the tab placement of the tabbed pane.

***31.21** (*Use tab panes*) Write a program using tab panes for performing integer and rational number arithmetic as shown in Figure 31.58.



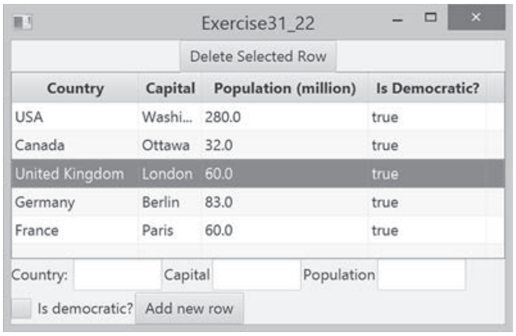
Number 1: 3 Number 2: 5 Result: 15.0

Number 1: 3 Number 2: 5 Result: 15.0

Figure 31.58 A tab pane is used to select panes that perform integer operations and rational number operations.

Sections 31.10

***31.22** (*Use table view*) Revise Listing 31.14 to add a button to delete the selected row from the table, as shown in Figure 31.59.



Country	Capital	Population (million)	Is Democratic?
USA	Washi...	280.0	true
Canada	Ottawa	32.0	true
United Kingdom	London	60.0	true
Germany	Berlin	83.0	true
France	Paris	60.0	true

Figure 31.59 Clicking the *Delete Selected Row* button removes the selected row from the table.

CHAPTER

32

Multithreading and Parallel Programming

Objectives

- To get an overview of multithreading (§32.2).
- To develop task classes by implementing the `Runnable` interface (§32.3).
- To create threads to run tasks using the `Thread` class (§32.3).
- To control threads using the methods in the `Thread` class (§32.4).
- To control animations using threads and use `Platform.runLater` to run the code in the application thread (§32.5).
- To execute tasks in a thread pool (§32.6).
- To use synchronized methods or blocks to synchronize threads to avoid race conditions (§32.7).
- To synchronize threads using locks (§32.8).
- To facilitate thread communications using conditions on locks (§§32.9 and 32.10).
- To use blocking queues (`ArrayBlockingQueue`, `LinkedBlockingQueue`, and `PriorityBlockingQueue`) to synchronize access to a queue (§32.11).
- To restrict the number of concurrent accesses to a shared resource using semaphores (§32.12).
- To use the resource-ordering technique to avoid deadlocks (§32.13).
- To describe the life cycle of a thread (§32.14).
- To create synchronized collections using the static methods in the `Collections` class (§32.15).
- To develop parallel programs using the Fork/Join Framework (§32.16).

32.1 Introduction

Multithreading enables multiple tasks in a program to be executed concurrently.

multithreading

One of the powerful features of Java is its built-in support for *multithreading*—the concurrent running of multiple tasks within a program. In many programming languages, you have to invoke system-dependent procedures and functions to implement multithreading. This chapter introduces the concepts of threads and how multithreading programs can be developed in Java.

32.2 Thread Concepts

A program may consist of many tasks that can run concurrently. A thread is the flow of execution, from beginning to end, of a task.

thread
task

A *thread* provides the mechanism for running a task. With Java, you can launch multiple threads from a program concurrently. These threads can be executed simultaneously in multiprocessor systems, as shown in Figure 32.1a.

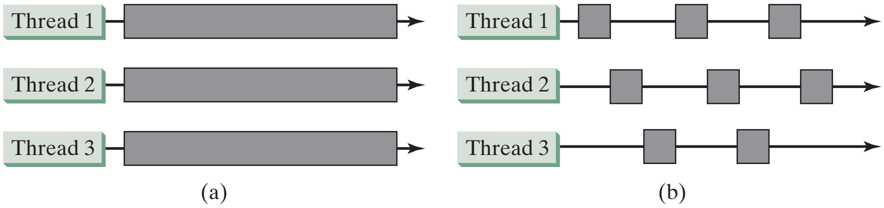


FIGURE 32.1 (a) Multiple threads running on multiple CPUs. (b) Multiple threads share a single CPU.

time sharing

In single-processor systems, as shown in Figure 32.1b, the multiple threads share CPU time, known as *time sharing*, and the operating system is responsible for scheduling and allocating resources to them. This arrangement is practical because most of the time the CPU is idle. It does nothing, for example, while waiting for the user to enter data.

Multithreading can make your program more responsive and interactive as well as enhance performance. For example, a good word processor lets you print or save a file while you are typing. In some cases, multithreaded programs run faster than single-threaded programs even on single-processor systems. Java provides exceptionally good support for creating and running threads, and for locking resources to prevent conflicts.

task
runnable object
thread

You can create additional threads to run concurrent tasks in the program. In Java, each task is an instance of the `Runnable` interface, also called a *runnable object*. A *thread* is essentially an object that facilitates the execution of a task.

32.2.1 Why is multithreading needed? How can multiple threads run simultaneously in a single-processor system?

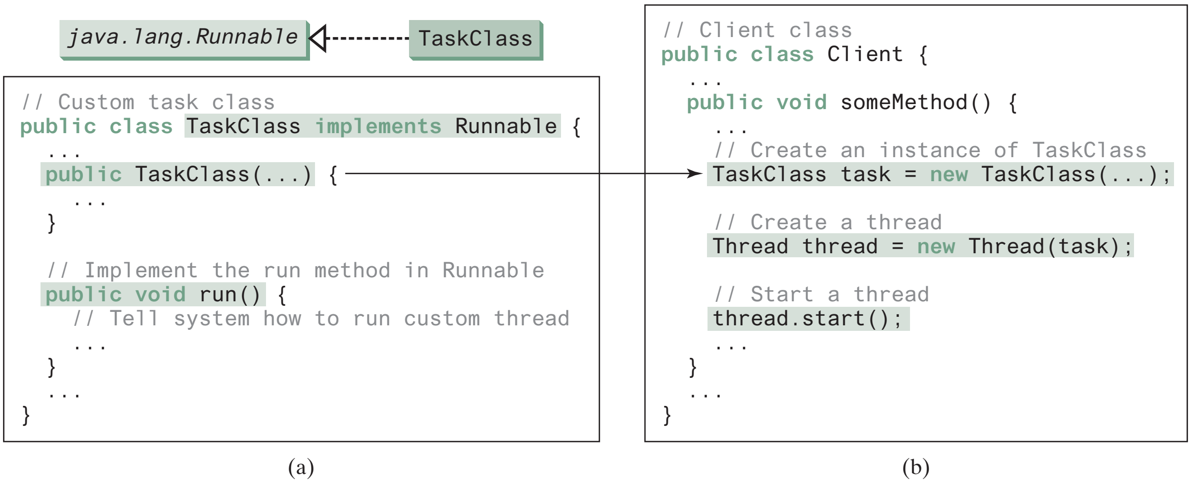
32.2.2 What is a runnable object? What is a thread?

32.3 Creating Tasks and Threads

A task class must implement the `Runnable` *interface. A task must be run from a thread.*

Runnable interface
run() method

Tasks are objects. To create tasks, you have to first define a class for tasks, which implements the `Runnable` interface. The `Runnable` interface is rather simple. All it contains is the `run()` method. You need to implement this method to tell the system how your thread is going to run. A template for developing a task class is shown in Figure 32.2a.



```
// Custom task class
public class TaskClass implements Runnable {
  ...
  public TaskClass(...) {
    ...
  }

  // Implement the run method in Runnable
  public void run() {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

(a)

```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create an instance of TaskClass
    TaskClass task = new TaskClass(...);

    // Create a thread
    Thread thread = new Thread(task);

    // Start a thread
    thread.start();
    ...
  }
  ...
}
```

(b)

FIGURE 32.2 Define a task class by implementing the Runnable interface.

Once you have defined a TaskClass, you can create a task using its constructor. For example,

```
TaskClass task = new TaskClass(...);
```

A task must be executed in a thread. The Thread class contains the constructors for creating threads and many useful methods for controlling threads. To create a thread for a task, use

Thread class

create a task

```
Thread thread = new Thread(task);
```

You can then invoke the start() method to tell the JVM that the thread is ready to run, as follows:

create a thread

```
thread.start();
```

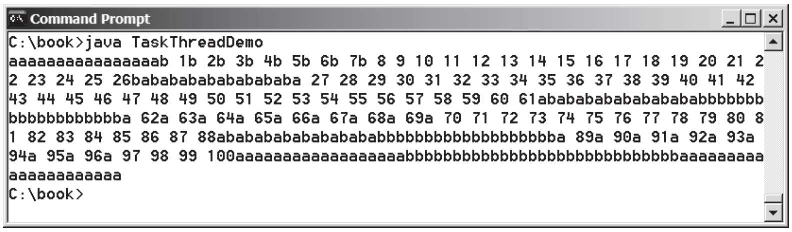
The JVM will execute the task by invoking the task's run() method. Figure 32.2b outlines the major steps for creating a task, a thread, and starting the thread.

start a thread

Listing 32.1 gives a program that creates three tasks and three threads to run them.

- The first task prints the letter *a* 100 times.
- The second task prints the letter *b* 100 times.
- The third task prints the integers 1 through 100.

When you run this program, the three threads will share the CPU and take turns printing letters and numbers on the console. Figure 32.3 shows a sample run of the program.



```
C:\book>java TaskThreadDemo
aaaaaaaaaaaaaaaab 1b 2b 3b 4b 5b 6b 7b 8 9 10 11 12 13 14 15 16 17 18 19 20 21 2
2 23 24 25 26bababababababababa 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61ababababababababab bbbbbb
bbbbbbbbbbbba 62a 63a 64a 65a 66a 67a 68a 69a 70 71 72 73 74 75 76 77 78 79 80 8
1 82 83 84 85 86 87 88ababababababababab bbbbbbbbbbbbbbbbba 89a 90a 91a 92a 93a
94a 95a 96a 97 98 99 100aaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaaaa
aaaaaaaaaaaa
C:\book>
```

FIGURE 32.3 Tasks printA, printB, and print100 are executed simultaneously to display the letter a 100 times, the letter b 100 times, and the numbers from 1 to 100.

LISTING 32.1 TaskThreadDemo.java

create tasks

create threads

start threads

task class

run

task class

run

```java
public class TaskThreadDemo {
  public static void main(String[] args) {
    // Create tasks
    Runnable printA = new PrintChar('a', 100);
    Runnable printB = new PrintChar('b', 100);
    Runnable print100 = new PrintNum(100);

    // Create threads
    Thread thread1 = new Thread(printA);
    Thread thread2 = new Thread(printB);
    Thread thread3 = new Thread(print100);

    // Start threads
    thread1.start();
    thread2.start();
    thread3.start();
  }
}

// The task for printing a character a specified number of times
class PrintChar implements Runnable {
  private char charToPrint; // The character to print
  private int times; // The number of times to repeat

  /** Construct a task with a specified character and number of
    * times to print the character
   */
  public PrintChar(char c, int t) {
    charToPrint = c;
    times = t;
  }

  @Override /** Override the run() method to tell the system
   * what task to perform
   */
  public void run() {
    for (int i = 0; i < times; i++) {
      System.out.print(charToPrint);
    }
  }
}

// The task class for printing numbers from 1 to n for a given n
class PrintNum implements Runnable {
  private int lastNum;

  /** Construct a task for printing 1, 2, ..., n */
  public PrintNum(int n) {
    lastNum = n;
  }

  @Override /** Tell the thread how to run */
  public void run() {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
    }
  }
}
```

The program creates three tasks (lines 4–6). To run them concurrently, three threads are created (lines 9–11). The `start()` method (lines 14–16) is invoked to start a thread that causes the `run()` method in the task to be executed. When the `run()` method completes, the thread terminates.

Because the first two tasks, `printA` and `printB`, have similar functionality, they can be defined in one task class `PrintChar` (lines 21–41). The `PrintChar` class implements `Runnable` and overrides the `run()` method (lines 36–40) with the print-character action. This class provides a framework for printing any single character a given number of times. The runnable objects, `printA` and `printB`, are instances of the `PrintChar` class.

The `PrintNum` class (lines 44–58) implements `Runnable` and overrides the `run()` method (lines 53–57) with the print-number action. This class provides a framework for printing numbers from 1 to *n*, for any integer *n*. The runnable object `print100` is an instance of the class `printNum` class.

Note

effect of concurrency

If you don't see the effect of these three threads running concurrently, increase the number of characters to be printed. For example, change line 4 to

```
Runnable printA = new PrintChar('a', 10000);
```

Important Note

run() method

The `run()` method in a task specifies how to perform the task. This method is automatically invoked by the JVM. You should not invoke it. Invoking `run()` directly merely executes this method in the same thread; no new thread is started.

Check Point

32.3.1 How do you define a task class? How do you create a thread for a task?

32.3.2 What would happen if you replace the `start()` method with the `run()` method in lines 14–16 in Listing 32.1?

```
print100.start();
printA.start();
printB.start();
```

Replaced by

```
print100.run();
printA.run();
printB.run();
```

32.3.3 What is wrong in the following two programs? Correct the errors.

```
public class Test implements Runnable {
  public static void main(String[] args) {
    new Test();
  }

  public Test() {
    Test task = new Test();
    new Thread(task).start();
  }

  public void run() {
    System.out.println("test");
  }
}
```

(a)

```
public class Test implements Runnable {
  public static void main(String[] args) {
    new Test();
  }

  public Test() {
    Thread t = new Thread(this);
    t.start();
    t.start();
  }

  public void run() {
    System.out.println("test");
  }
}
```

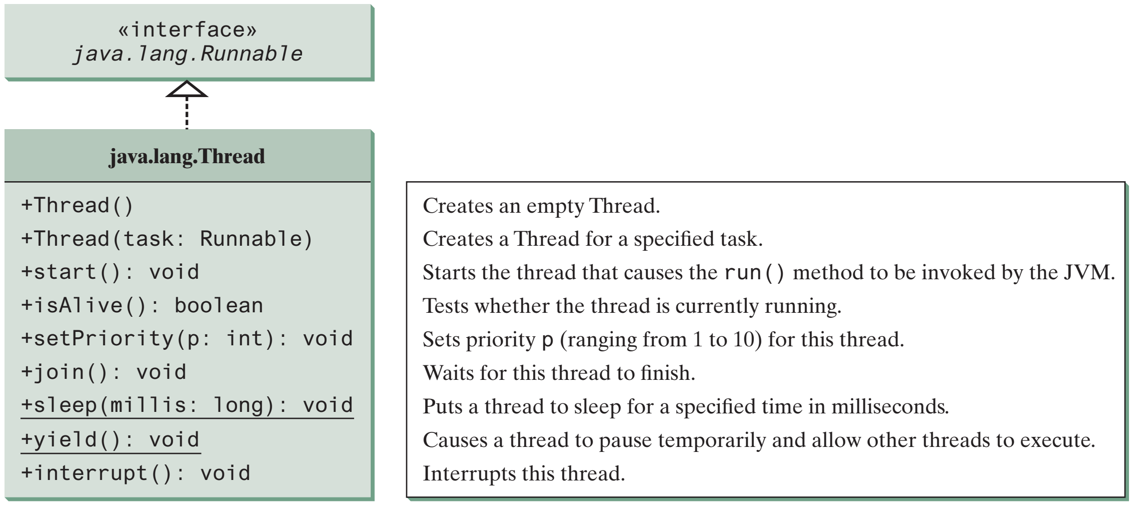
(b)

32.4 The Thread Class

Key Point

*The **Thread** class contains the constructors for creating threads for tasks and the methods for controlling threads.*

Figure 32.4 shows the class diagram for the **Thread** class.



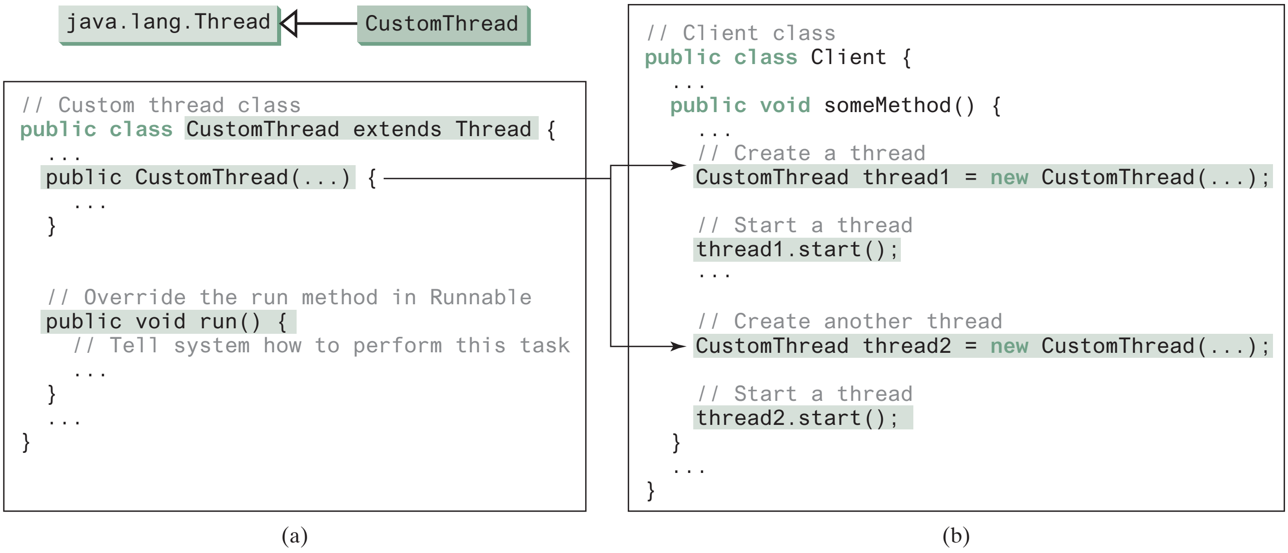
java.lang.Thread	
+Thread()	Creates an empty Thread.
+Thread(task: Runnable)	Creates a Thread for a specified task.
+start(): void	Starts the thread that causes the run() method to be invoked by the JVM.
+isAlive(): boolean	Tests whether the thread is currently running.
+setPriority(p: int): void	Sets priority p (ranging from 1 to 10) for this thread.
+join(): void	Waits for this thread to finish.
+sleep(millis: long): void	Puts a thread to sleep for a specified time in milliseconds.
+yield(): void	Causes a thread to pause temporarily and allow other threads to execute.
+interrupt(): void	Interrupts this thread.

FIGURE 32.4 The **Thread** class contains the methods for controlling threads.

Note

Since the **Thread** class implements **Runnable**, you could define a class that extends **Thread** and implements the **run** method, as shown in Figure 32.5a, then create an object from the class and invoke its **start** method in a client program to start the thread, as shown in Figure 32.5b.

separating task from thread



```
// Custom thread class
public class CustomThread extends Thread {
  ...
  public CustomThread(...) {
    ...
  }

  // Override the run method in Runnable
  public void run() {
    // Tell system how to perform this task
    ...
  }
  ...
}
```

```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create a thread
    CustomThread thread1 = new CustomThread(...);

    // Start a thread
    thread1.start();
    ...

    // Create another thread
    CustomThread thread2 = new CustomThread(...);

    // Start a thread
    thread2.start();
  }
  ...
}
```

FIGURE 32.5 Define a thread class by extending the **Thread** class.

This approach is, however, not recommended because it mixes the task and the mechanism of running the task. Separating the task from the thread is a preferred design.

Note

The Thread class also contains the stop(), suspend(), and resume() methods. As of Java 2, these methods were *deprecated* (or *outdated*) because they are known to be inherently unsafe. Instead of using the stop() method, you should assign null to a Thread variable to indicate that it has stopped.

deprecated method

You can use the yield() method to temporarily release time for other threads. For example, suppose that you modify the code in the run() method in lines 53–57 for PrintNum in Listing 32.1 as follows:

yield()

```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    Thread.yield();
  }
}
```

Every time a number is printed, the thread of the print100 task is yielded to other threads.

The sleep(long millis) method puts the thread to sleep for a specified time in milliseconds to allow other threads to execute. For example, suppose that you modify the code in lines 53–57 in Listing 32.1 as follows:

sleep(long)

```
public void run() {
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
      if (i >= 50) Thread.sleep(1);
    }
  }
  catch (InterruptedException ex) {
  }
}
```

Every time a number (>= 50) is printed, the thread of the print100 task is put to sleep for 1 millisecond.

The sleep method may throw an InterruptedException, which is a checked exception. Such an exception may occur when a sleeping thread's interrupt() method is called. The interrupt() method is very rarely invoked on a thread, so an InterruptedException is unlikely to occur. But since Java forces you to catch checked exceptions, you have to put it in a try-catch block. If a sleep method is invoked in a loop, you should wrap the loop in a try-catch block, as shown in (a) below. If the loop is outside the try-catch block, as shown in (b), the thread may continue to execute even though it is being interrupted.

InterruptedException

```
public void run() {
  try {
    while (...) {
      ...
      Thread.sleep(1000);
    }
  }
  catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
```

(a) Correct

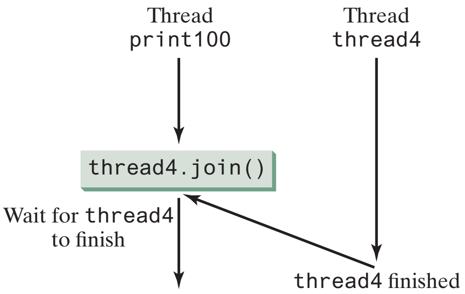
```
public void run() {
  while (...) {
    try {
      ...
      Thread.sleep(sleepTime);
    }
    catch (InterruptedException ex) {
      ex.printStackTrace();
    }
  }
}
```

(b) Incorrect

join()

You can use the `join()` method to force one thread to wait for another thread to finish. For example, suppose that you modify the code in lines 53–57 in Listing 32.1 as follows:

```
public void run() {
  Thread thread4 = new Thread(
    new PrintChar('c', 40));
  thread4.start();
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print (" " + i);
      if (i == 50) thread4.join();
    }
  }
  catch (InterruptedException ex) {
  }
}
```



A new `thread4` is created and it prints character *c* 40 times. The numbers from `50` to `100` are printed after thread `thread4` is finished.

Java assigns every thread a priority. By default, a thread inherits the priority of the thread that spawned it. You can increase or decrease the priority of any thread by using the `setPriority` method and you can get the thread's priority by using the `getPriority` method. Priorities are numbers ranging from `1` to `10`. The `Thread` class has the `int` constants `MIN_PRIORITY`, `NORM_PRIORITY`, and `MAX_PRIORITY`, representing `1`, `5`, and `10`, respectively. The priority of the main thread is `Thread.NORM_PRIORITY`.

setPriority(int)

The JVM always picks the currently runnable thread with the highest priority. A lower priority thread can run only when no higher priority threads are running. If all runnable threads have equal priorities, each is assigned an equal portion of the CPU time in a circular queue. This is called *round-robin scheduling*. For example, suppose that you insert the following code in line 16 in Listing 32.1:

round-robin scheduling

```
thread3.setPriority(Thread.MAX_PRIORITY);
```

The thread for the `print100` task will be finished first.

Tip

The priority numbers may be changed in a future version of Java. To minimize the impact of any changes, use the constants in the `Thread` class to specify thread priorities.

Tip

A thread may never get a chance to run if there is always a higher priority thread running or a same-priority thread that never yields. This situation is known as *contention* or *starvation*. To avoid contention, the thread with higher priority must periodically invoke the `sleep` or `yield` method to give a thread with a lower or the same priority a chance to run.

contention or starvation

Check Point

32.4.1 Which of the following methods are instance methods in `java.lang.Thread`? Which method may throw an `InterruptedException`? Which of them are deprecated in Java?

`run`, `start`, `stop`, `suspend`, `resume`, `sleep`, `interrupt`, `yield`, `join`

32.4.2 If a loop contains a method that throws an `InterruptedException`, why should the loop be placed inside a `try-catch` block?

32.4.3 How do you set a priority for a thread? What is the default priority?

32.5 Animation Using Threads and the Platform.runLater Method

Key Point

You can use a thread to control an animation and run the code in JavaFX GUI thread using the `Platform.runLater` *method.*

The use of a `Timeline` object to control animations was introduced in Section 15.11, Animation. Alternatively, you can also use a thread to control animation. Listing 32.2 gives an example that displays flashing text on a label, as shown in Figure 32.6.

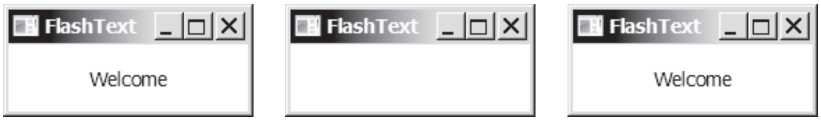


Figure 32.6 The text “Welcome” blinks.

Listing 32.2 FlashText.java

```
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class FlashText extends Application {
  private String text = "";

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    StackPane pane = new StackPane();
    Label lblText = new Label("Programming is fun");
    pane.getChildren().add(lblText);

    new Thread(new Runnable() {
      @Override
      public void run() {
        try {
          while (true) {
            if (lblText.getText().trim().length() == 0)
              text = "Welcome";
            else
              text = "";

            Platform.runLater(new Runnable() { // Run from JavaFX GUI
              @Override
              public void run() {
                lblText.setText(text);
              }
            });

            Thread.sleep(200);
          }
        }
        catch (InterruptedException ex) {
        }
      }
    }).start();

```

create a label

label in a pane

create a thread

run thread

change text

Platform.runLater

update GUI

sleep

```
42       // Create a scene and place it in the stage
43       Scene scene = new Scene(pane, 200, 50);
44       primaryStage.setTitle("FlashText"); // Set the stage title
45       primaryStage.setScene(scene); // Place the scene in the stage
46       primaryStage.show(); // Display the stage
47     }
48   }
```

The program creates a Runnable object in an anonymous inner class (lines 17–40). This object is started in line 40 and runs continuously to change the text in the label. It sets a text in the label if the label is blank (line 23) and sets its text blank (line 25) if the label has a text. The text is set and unset to simulate a flashing effect.

JavaFX application thread

JavaFX GUI is run from the *JavaFX application thread*. The flashing control is run from a separate thread. The code in a nonapplication thread cannot update GUI in the application thread. To update the text in the label, a new Runnable object is created in lines 27–32. Invoking Platform.runLater(Runnable r) tells the system to run this Runnable object in the application thread.

Platform.runLater

The anonymous inner classes in this program can be simplifed using lambda expressions as follows:

```
new Thread(() -> { // lambda expression
  try {
    while (true) {
      if (lblText.getText().trim().length() == 0)
        text = "Welcome";
      else
        text = "";

      Platform.runLater(() -> lblText.setText(text)); // lambda exp

      Thread.sleep(200);
    }
  }
  catch (InterruptedException ex) {
  }
}).start();
```

Check Point

32.5.1 What causes the text to flash?

32.5.2 Is an instance of FlashText a runnable object?

32.5.3 What is the purpose of using Platform.runLater?

32.5.4 Can you replace the code in lines 27–32 using the following code?

```
Platform.runLater(e -> lblText.setText(text));
```

32.5.5 What happens if line 34 (Thread.sleep(200)) is not used?

32.5.6 There is an issue in Listing 16.9, ListViewDemo. If you press the CTRL key and select Canada, Demark, and China in this order, you will get an ArrayIndexOutBoundsException. What is the reason for this error and how do you fix it? (Thanks to Henri Heimonen of Finland for contributing this question).

Key Point

32.6 Thread Pools

A thread pool can be used to execute tasks efficiently.

In Section 32.3, Creating Tasks and Threads, you learned how to define a task class by implementing java.lang.Runnable, and how to create a thread to run a task like this:

```
Runnable task = new TaskClass(...);
new Thread(task).start();
```

This approach is convenient for a single task execution, but it is not efficient for a large number of tasks because you have to create a thread for each task. Starting a new thread for each task could limit throughput and cause poor performance. Using a *thread pool* is an ideal way to manage the number of tasks executing concurrently. Java provides the `Executor` interface for executing tasks in a thread pool and the `ExecutorService` interface for managing and controlling tasks. `ExecutorService` is a subinterface of `Executor`, as shown in Figure 32.7.

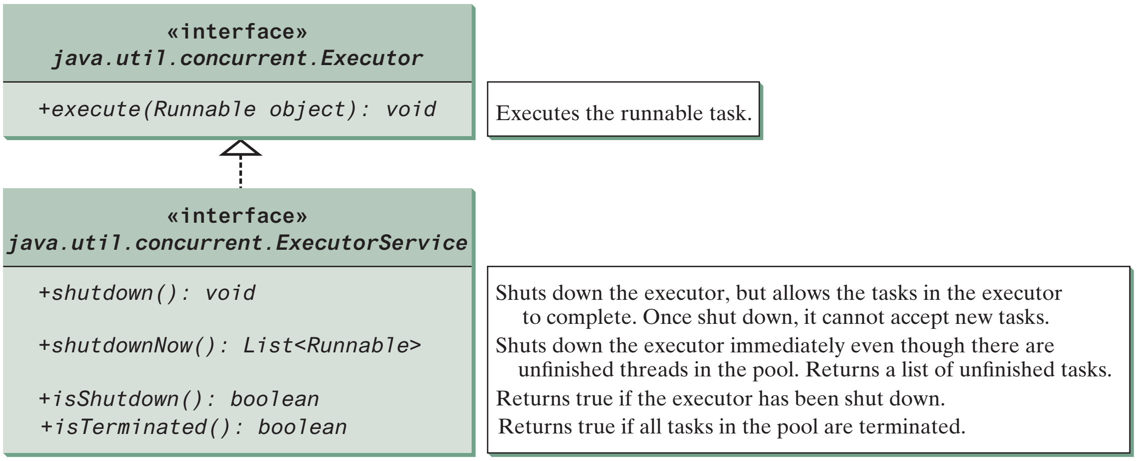

«interface»
java.util.concurrent.Executor
+execute(Runnable object): void
Executes the runnable task.
«interface»
java.util.concurrent.ExecutorService
+shutdown(): void
+shutdownNow(): List<Runnable>
+isShutdown(): boolean
+isTerminated(): boolean
Shuts down the executor, but allows the tasks in the executor to complete. Once shut down, it cannot accept new tasks.
Shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.
Returns true if the executor has been shut down.
Returns true if all tasks in the pool are terminated.

FIGURE 32.7 The `Executor` interface executes threads and the `ExecutorService` subinterface manages threads.

To create an `Executor` object, use the static methods in the `Executors` class, as shown in Figure 32.8. The `newFixedThreadPool(int)` method creates a fixed number of threads in a pool. If a thread completes executing a task, it can be reused to execute another task. If a thread terminates due to a failure prior to shutdown, a new thread will be created to replace it if all the threads in the pool are not idle and there are tasks waiting for execution. The `newCachedThreadPool()` method creates a new thread if all the threads in the pool are not idle and there are tasks waiting for execution. A thread in a cached pool will be terminated if it has not been used for 60 seconds. A cached pool is efficient for many short tasks.

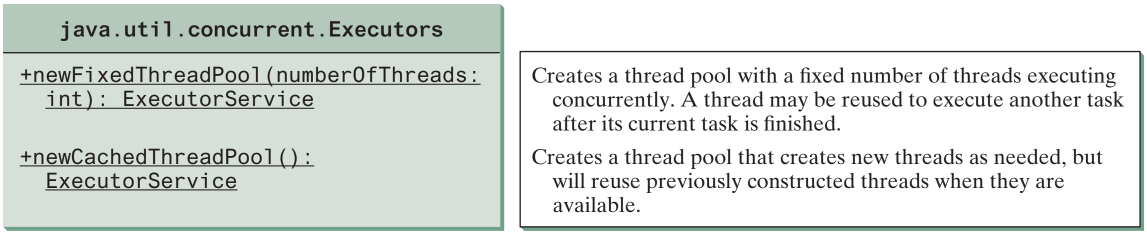

java.util.concurrent.Executors
+newFixedThreadPool(numberOfThreads: int): ExecutorService
+newCachedThreadPool(): ExecutorService
Creates a thread pool with a fixed number of threads executing concurrently. A thread may be reused to execute another task after its current task is finished.
Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

FIGURE 32.8 The `Executors` class provides static methods for creating `Executor` objects.

Listing 32.3 shows how to rewrite Listing 32.1 using a thread pool.

LISTING 32.3 ExecutorDemo.java

```
import java.util.concurrent.*;

public class ExecutorDemo {
```

create executor

submit task

shut down executor

```
 4     public static void main(String[] args) {
 5       // Create a fixed thread pool with maximum three threads
 6       ExecutorService executor = Executors.newFixedThreadPool(3);
 7 
 8       // Submit runnable tasks to the executor
 9       executor.execute(new PrintChar('a', 100));
10       executor.execute(new PrintChar('b', 100));
11       executor.execute(new PrintNum(100));
12 
13       // Shut down the executor
14       executor.shutdown();
15     }
16   }
```

Line 6 creates a thread pool executor with a total of three threads maximum. Classes `PrintChar` and `PrintNum` are defined in Listing 32.1. Line 9 creates a task, `new PrintChar('a', 100)`, and adds it to the pool. Similarly, another two runnable tasks are created and added to the same pool in lines 10 and 11. The executor creates three threads to execute three tasks concurrently.

Suppose you replace line 6 with

```
ExecutorService executor = Executors.newFixedThreadPool(1);
```

What will happen? The three runnable tasks will be executed sequentially because there is only one thread in the pool.

Suppose you replace line 6 with

```
ExecutorService executor = Executors.newCachedThreadPool();
```

What will happen? New threads will be created for each waiting task, so all the tasks will be executed concurrently.

The `shutdown()` method in line 14 tells the executor to shut down. No new tasks can be accepted, but any existing tasks will continue to finish.

Tip

If you need to create a thread for just one task, use the `Thread` class. If you need to create threads for multiple tasks, it is better to use a thread pool.

Check Point

32.6.1 What are the benefits of using a thread pool?

32.6.2 How do you create a thread pool with three fixed threads? How do you submit a task to a thread pool? How do you know that all the tasks are finished?

32.7 Thread Synchronization

Key Point

Thread synchronization is to coordinate the execution of the dependent threads.

A shared resource may become corrupted if it is accessed simultaneously by multiple threads. The following example demonstrates the problem.

Suppose that you create and launch 100 threads, each of which adds a penny to an account. Define a class named `Account` to model the account, a class named `AddAPennyTask` to add a penny to the account, and a main class that creates and launches threads. The relationships of these classes are shown in Figure 32.9. The program is given in Listing 32.4.

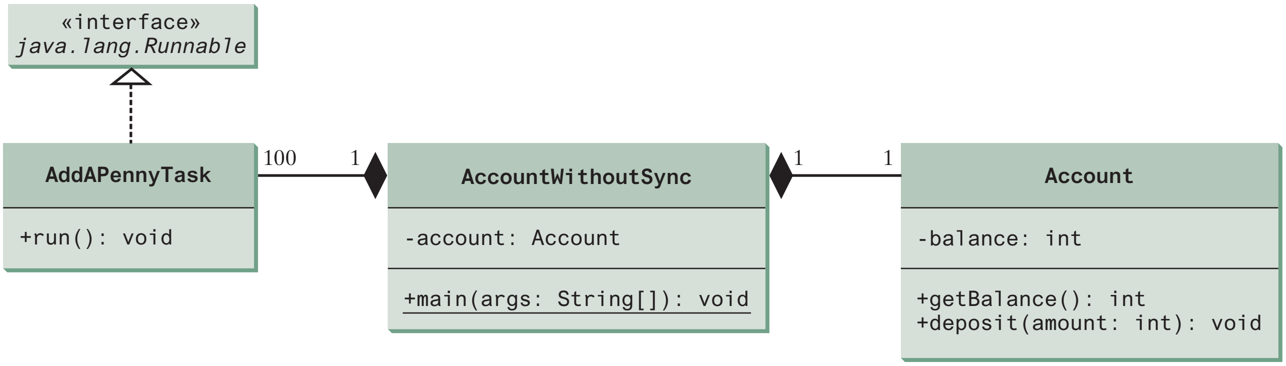


Figure 32.9 AccountWithoutSync contains an instance of Account and 100 threads of AddAPennyTask.

Listing 32.4 AccountWithoutSync.java

```
import java.util.concurrent.*;

public class AccountWithoutSync {
  private static Account account = new Account();

  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();

    // Create and launch 100 threads
    for (int i = 0; i < 100; i++) {
      executor.execute(new AddAPennyTask());
    }

    executor.shutdown();

    // Wait until all tasks are finished
    while (!executor.isTerminated()) {
    }

    System.out.println("What is balance? " + account.getBalance());
  }

  // A thread for adding a penny to the account
  private static class AddAPennyTask implements Runnable {
    public void run() {
      account.deposit(1);
    }
  }

  // An inner class for account
  private static class Account {
    private int balance = 0;

    public int getBalance() {
      return balance;
    }

    public void deposit(int amount) {
      int newBalance = balance + amount;

      // This delay is deliberately added to magnify the
```

create executor

submit task

shut down executor

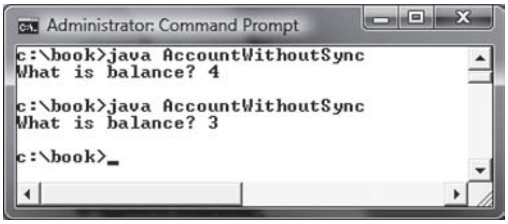
wait for all tasks to terminate

```
42          // data-corruption problem and make it easy to see.
43          try {
44            Thread.sleep(5);
45          }
46          catch (InterruptedException ex) {
47          }
48
49          balance = newBalance;
50        }
51      }
52    }
```

The classes AddAPennyTask and Account in lines 24–51 are inner classes. Line 4 creates an Account with initial balance 0. Line 11 creates a task to add a penny to the account and submits the task to the executor. Line 11 is repeated 100 times in lines 10–12. The program repeatedly checks whether all tasks are completed in lines 17 and 18. The account balance is displayed in line 20 after all tasks are completed.

The program creates 100 threads executed in a thread pool executor (lines 10–12). The isTerminated() method (line 17) is used to test whether all the threads in the pool are terminated.

The balance of the account is initially 0 (line 32). When all the threads are finished, the balance should be 100 but the output is unpredictable. As can be seen in Figure 32.10, the answers are wrong in the sample run. This demonstrates the data-corruption problem that occurs when all the threads have access to the same data source simultaneously.



```
c:\book>java AccountWithoutSync
What is balance? 4

c:\book>java AccountWithoutSync
What is balance? 3

c:\book>_
```

FIGURE 32.10 The AccountWithoutSync program causes data inconsistency.

Lines 39–49 could be replaced by one statement:

```
balance = balance + amount;
```

It is highly unlikely, although plausible, that the problem can be replicated using this single statement. The statements in lines 39–49 are deliberately designed to magnify the data-corruption problem and make it easy to see. If you run the program several times but still do not see the problem, increase the sleep time in line 44. This will increase the chances for showing the problem of data inconsistency.

What, then, caused the error in this program? A possible scenario is shown in Figure 32.11.

Step	Balance	Task 1	Task 2
1	0	`newBalance = balance + 1;`	
2	0		`newBalance = balance + 1;`
3	1	`balance = newBalance;`	
4	1		`balance = newBalance;`

FIGURE 32.11 Task 1 and Task 2 both add 1 to the same balance.

In Step 1, Task 1 gets the balance from the account. In Step 2, Task 2 gets the same balance from the account. In Step 3, Task 1 writes a new balance to the account. In Step 4, Task 2 writes a new balance to the account.

The effect of this scenario is that Task 1 does nothing because in Step 4, Task 2 overrides Task 1's result. Obviously, the problem is that Task 1 and Task 2 are accessing a common resource in a way that causes a conflict. This is a common problem, known as a *race condition*, in multithreaded programs. A class is said to be *thread-safe* if an object of the class does not cause a race condition in the presence of multiple threads. As demonstrated in the preceding example, the `Account` class is not thread-safe.

race condition

thread-safe

32.7.1 The `synchronized` Keyword

To avoid race conditions, it is necessary to prevent more than one thread from simultaneously entering a certain part of the program, known as the *critical region*. The critical region in Listing 32.4 is the entire `deposit` method. You can use the keyword `synchronized` to synchronize the method so that only one thread can access the method at a time. There are several ways to correct the problem in Listing 32.4. One approach is to make `Account` thread-safe by adding the keyword `synchronized` in the `deposit` method in line 38, as follows:

critical region

```
public synchronized void deposit(double amount)
```

A synchronized method acquires a lock before it executes. A lock is a mechanism for exclusive use of a resource. In the case of an instance method, the lock is on the object for which the method was invoked. In the case of a static method, the lock is on the class. If one thread invokes a synchronized instance method (respectively, static method) on an object, the lock of that object (respectively, class) is acquired first, then the method is executed, and finally the lock is released. Another thread invoking the same method of that object (respectively, class) is blocked until the lock is released.

With the `deposit` method synchronized, the preceding scenario cannot happen. If Task 1 enters the method, Task 2 is blocked until Task 1 finishes the method, as shown in Figure 32.12.

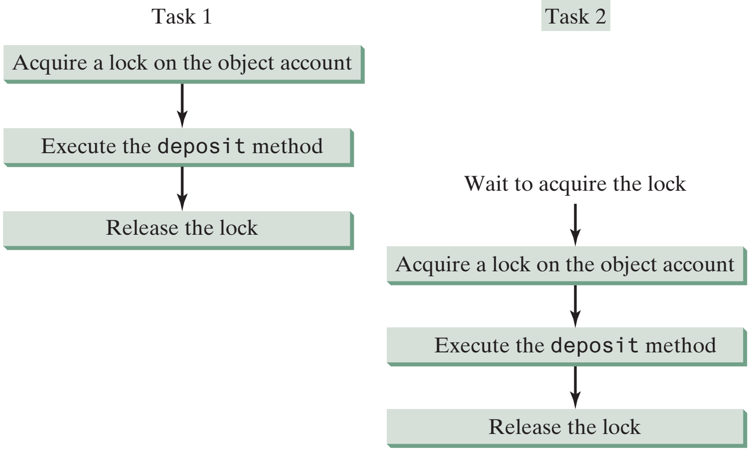

Task 1
Acquire a lock on the object account
Execute the deposit method
Release the lock
Task 2
Wait to acquire the lock
Acquire a lock on the object account
Execute the deposit method
Release the lock

FIGURE 32.12 Task 1 and Task 2 are synchronized.

32.7.2 Synchronizing Statements

Invoking a synchronized instance method of an object acquires a lock on the object, and invoking a synchronized static method of a class acquires a lock on the class. A synchronized statement can be used to acquire a lock on any object, not just *this* object, when executing a block synchronized block

of the code in a method. This block is referred to as a *synchronized block*. The general form of a synchronized statement is as follows:

```
synchronized (expr) {
  statements;
}
```

The expression `expr` must evaluate to an object reference. If the object is already locked by another thread, the thread is blocked until the lock is released. When a lock is obtained on the object, the statements in the synchronized block are executed and then the lock is released.

Synchronized statements enable you to synchronize part of the code in a method instead of the entire method. This increases concurrency. You can make Listing 32.4 thread-safe by placing the statement in line 26 inside a synchronized block:

```
synchronized (account) {
  account.deposit(1);
}
```

Note

Any synchronized instance method can be converted into a synchronized statement. For example, the following synchronized instance method in (a) is equivalent to (b):

```
public synchronized void xMethod() {
  // method body
}
```

(a)

```
public void xMethod() {
  synchronized (this) {
    // method body
  }
}
```

(b)

Check Point

32.7.1 Give some examples of possible resource corruption when running multiple threads. How do you synchronize conflicting threads?

32.7.2 Suppose you place the statement in line 26 of Listing 32.4 inside a synchronized block to avoid race conditions, as follows:

```
synchronized (this) {
  account.deposit(1);
}
```

Will it work?

32.8 Synchronization Using Locks

Key Point

Locks and conditions can be explicitly used to synchronize threads.

Recall that in Listing 32.4, 100 tasks deposit a penny to the same account concurrently, which causes conflicts. To avoid it, you use the `synchronized` keyword in the `deposit` method, as follows:

```
public synchronized void deposit(double amount)
```

lock

A synchronized instance method implicitly acquires a *lock* on the instance before it executes the method.

Java enables you to acquire locks explicitly, which give you more control for coordinating threads. A lock is an instance of the `Lock` interface, which defines the methods for acquiring and releasing locks, as shown in Figure 32.13. A lock may also use the `newCondition()` method to create any number of `Condition` objects, which can be used for thread communications.

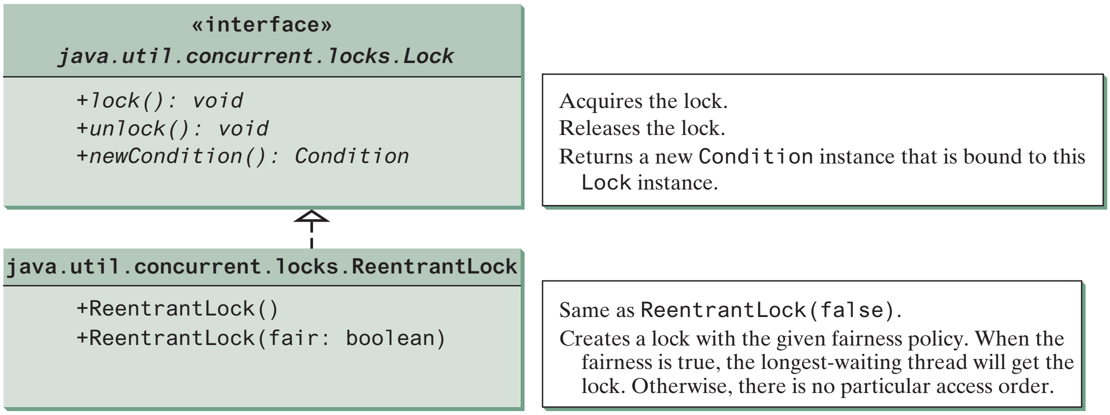

«interface»
java.util.concurrent.locks.Lock
+lock(): void
+unlock(): void
+newCondition(): Condition
Acquires the lock.
Releases the lock.
Returns a new Condition instance that is bound to this Lock instance.
java.util.concurrent.locks.ReentrantLock
+ReentrantLock()
+ReentrantLock(fair: boolean)
Same as ReentrantLock(false).
Creates a lock with the given fairness policy. When the fairness is true, the longest-waiting thread will get the lock. Otherwise, there is no particular access order.

FIGURE 32.13 The ReentrantLock class implements the Lock interface to represent a lock.

ReentrantLock is a concrete implementation of Lock for creating mutually exclusive locks. You can create a lock with the specified *fairness policy*. True fairness policies guarantee that the longest waiting thread will obtain the lock first. False fairness policies grant a lock to a waiting thread arbitrarily. Programs using fair locks accessed by many threads may have poorer overall performance than those using the default setting, but they have smaller variances in times to obtain locks and prevent starvation.

fairness policy

Listing 32.5 revises the program in Listing 32.7 to synchronize the account modification using explicit locks.

LISTING 32.5 AccountWithSyncUsingLock.java

package for locks

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class AccountWithSyncUsingLock {
  private static Account account = new Account();

  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();

    // Create and launch 100 threads
    for (int i = 0; i < 100; i++) {
      executor.execute(new AddAPennyTask());
    }

    executor.shutdown();

    // Wait until all tasks are finished
    while (!executor.isTerminated()) {
    }

    System.out.println("What is balance? " + account.getBalance());
  }

  // A thread for adding a penny to the account
  public static class AddAPennyTask implements Runnable {
    public void run() {
      account.deposit(1);
    }
  }

```

create a lock

acquire the lock

release the lock

```
31    // An inner class for Account
32    public static class Account {
33      private static Lock lock = new ReentrantLock(); // Create a lock
34      private int balance = 0;
35
36      public int getBalance() {
37        return balance;
38      }
39
40      public void deposit(int amount) {
41        lock.lock(); // Acquire the lock
42
43        try {
44          int newBalance = balance + amount;
45
46          // This delay is deliberately added to magnify the
47          // data-corruption problem and make it easy to see.
48          Thread.sleep(5);
49
50          balance = newBalance;
51        }
52        catch (InterruptedException ex) {
53        }
54        finally {
55          lock.unlock(); // Release the lock
56        }
57      }
58    }
59  }
```

Line 33 creates a lock, line 41 acquires the lock, and line 55 releases the lock.

Tip

It is a good practice to always immediately follow a call to `lock()` with a `try-catch` block and release the lock in the `finally` clause, as shown in lines 41–56, to ensure that the lock is always released.

Listing 32.5 can be implemented using a synchronize method for `deposit` rather than using a lock. In general, using `synchronized` methods or statements is simpler than using explicit locks for mutual exclusion. However, using explicit locks is more intuitive and flexible to synchronize threads with conditions, as you will see in the next section.

Check Point

32.8.1 How do you create a lock object? How do you acquire a lock and release a lock?

32.9 Cooperation among Threads

Key Point

Conditions on locks can be used to coordinate thread interactions.

condition

Thread synchronization suffices to avoid race conditions by ensuring the mutual exclusion of multiple threads in the critical region, but sometimes you also need a way for threads to cooperate. *Conditions* can be used to facilitate communications among threads. A thread can specify what to do under a certain condition. Conditions are objects created by invoking the `newCondition()` method on a `Lock` object. Once a condition is created, you can use its `await()`, `signal()`, and `signalAll()` methods for thread communications, as shown in Figure 32.14. The `await()` method causes the current thread to wait until the condition is signaled. The `signal()` method wakes up one waiting thread, and the `signalAll()` method wakes all waiting threads.

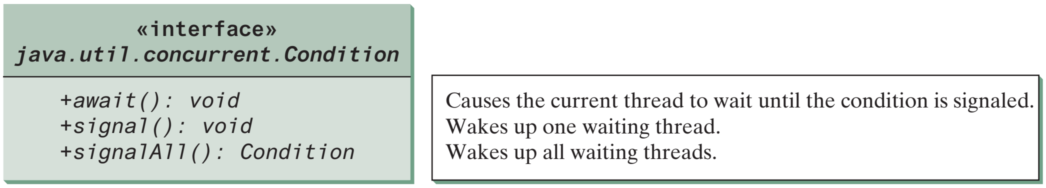

«interface»
java.util.concurrent.Condition
+await(): void
+signal(): void
+signalAll(): Condition
Causes the current thread to wait until the condition is signaled.
Wakes up one waiting thread.
Wakes up all waiting threads.

FIGURE 32.14 The `Condition` interface defines the methods for performing synchronization.

thread cooperation example

Let us use an example to demonstrate thread communications. Suppose you create and launch two tasks: one that deposits into an account, and one that withdraws from the same account. The withdraw task has to wait if the amount to be withdrawn is more than the current balance. Whenever new funds are deposited into the account, the deposit task notifies the withdraw thread to resume. If the amount is still not enough for a withdrawal, the withdraw thread has to continue to wait for a new deposit.

To synchronize the operations, use a lock with a condition: `newDeposit` (i.e., new deposit added to the account). If the balance is less than the amount to be withdrawn, the withdraw task will wait for the `newDeposit` condition. When the deposit task adds money to the account, the task signals the waiting withdraw task to try again. The interaction between the two tasks is shown in Figure 32.15.

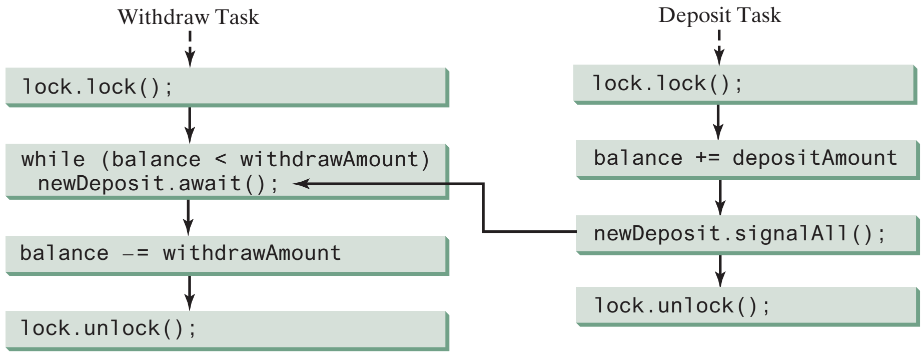
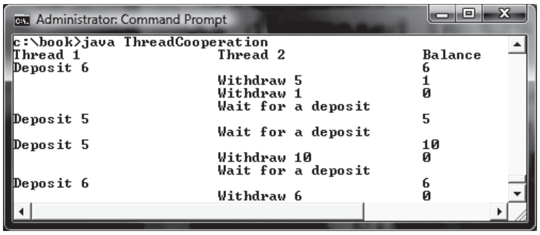


FIGURE 32.15 The condition `newDeposit` is used for communications between the two threads.

You create a condition from a `Lock` object. To use a condition, you have to first obtain a lock. The `await()` method causes the thread to wait and automatically releases the lock on the condition. Once the condition is right, the thread reacquires the lock and continues executing.

Assume the initial balance is `0` and the amount to deposit and withdraw are randomly generated. Listing 32.6 gives the program. A sample run of the program is shown in Figure 32.16.



```
c:\book>java ThreadCooperation
Thread 1                 Thread 2                      Balance
Deposit 6                                              6
                         Withdraw 5                    1
                         Withdraw 1                    0
                         Wait for a deposit
Deposit 5                                              5
                         Wait for a deposit
Deposit 5                                              10
                         Withdraw 10                   0
                         Wait for a deposit
Deposit 6                                              6
                         Withdraw 6                    0
```

FIGURE 32.16 The withdraw task waits if there are not sufficient funds to withdraw.

Listing 32.6 ThreadCooperation.java

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class ThreadCooperation {
  private static Account account = new Account();

  public static void main(String[] args) {
    // Create a thread pool with two threads
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new DepositTask());
    executor.execute(new WithdrawTask());
    executor.shutdown();

    System.out.println("Thread 1\t\tThread 2\t\tBalance");
  }

  public static class DepositTask implements Runnable {
    @Override // Keep adding an amount to the account
    public void run() {
      try { // Purposely delay it to let the withdraw method proceed
        while (true) {
          account.deposit((int)(Math.random() * 10) + 1);
          Thread.sleep(1000);
        }
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }

  public static class WithdrawTask implements Runnable {
    @Override // Keep subtracting an amount from the account
    public void run() {
      while (true) {
        account.withdraw((int)(Math.random() * 10) + 1);
      }
    }
  }

  // An inner class for account
  private static class Account {
    // Create a new lock
    private static Lock lock = new ReentrantLock();

    // Create a condition
    private static Condition newDeposit = lock.newCondition();

    private int balance = 0;

    public int getBalance() {
      return balance;
    }

    public void withdraw(int amount) {
      lock.lock(); // Acquire the lock
      try {
        while (balance < amount) {
```

create two threads

create a lock

create a condition

acquire the lock

```
59            System.out.println("\t\t\tWait for a deposit");
60            newDeposit.await();
61          }
62  
63          balance -= amount;
64          System.out.println("\t\t\tWithdraw " + amount +
65            "\t\t" + getBalance());
66        }
67        catch (InterruptedException ex) {
68          ex.printStackTrace();
69        }
70        finally {
71          lock.unlock(); // Release the lock
72        }
73      }
74  
75      public void deposit(int amount) {
76        lock.lock(); // Acquire the lock
77        try {
78          balance += amount;
79          System.out.println("Deposit " + amount +
80            "\t\t\t\t\t" + getBalance());
81  
82          // Signal thread waiting on the condition
83          newDeposit.signalAll();
84        }
85        finally {
86          lock.unlock(); // Release the lock
87        }
88      }
89    }
90  }
```

wait on the condition

release the lock

acquire the lock

signal threads

release the lock

The example creates a new inner class named Account to model the account with two methods, deposit(int) and withdraw(int), a class named DepositTask to add an amount to the balance, a class named WithdrawTask to withdraw an amount from the balance, and a main class that creates and launches two threads.

The program creates and submits the deposit task (line 10) and the withdraw task (line 11). The deposit task is purposely put to sleep (line 23) to let the withdraw task run. When there are not enough funds to withdraw, the withdraw task waits (line 59) for notification of the balance change from the deposit task (line 83).

A lock is created in line 44. A condition named newDeposit on the lock is created in line 47. A condition is bound to a lock. Before waiting or signaling the condition, a thread must first acquire the lock for the condition. The withdraw task acquires the lock in line 56, waits for the newDeposit condition (line 60) when there is not a sufficient amount to withdraw, and releases the lock in line 71. The deposit task acquires the lock in line 76 and signals all waiting threads (line 83) for the newDeposit condition after a new deposit is made.

What will happen if you replace the while loop in lines 58–61 with the following if statement?

```
if (balance < amount) {
  System.out.println("\t\t\tWait for a deposit");
  newDeposit.await();
}
```

The deposit task will notify the withdraw task whenever the balance changes. (balance < amount) may still be true when the withdraw task is awakened. Using the if statement would lead to incorrect withdraw. Using the loop statement, the withdraw task will have a chance to recheck the condition.

ever-waiting threads

Caution

Once a thread invokes `await()` on a condition, the thread waits for a signal to resume. If you forget to call `signal()` or `signalAll()` on the condition, the thread will wait forever.

IllegalMonitorStateException

Caution

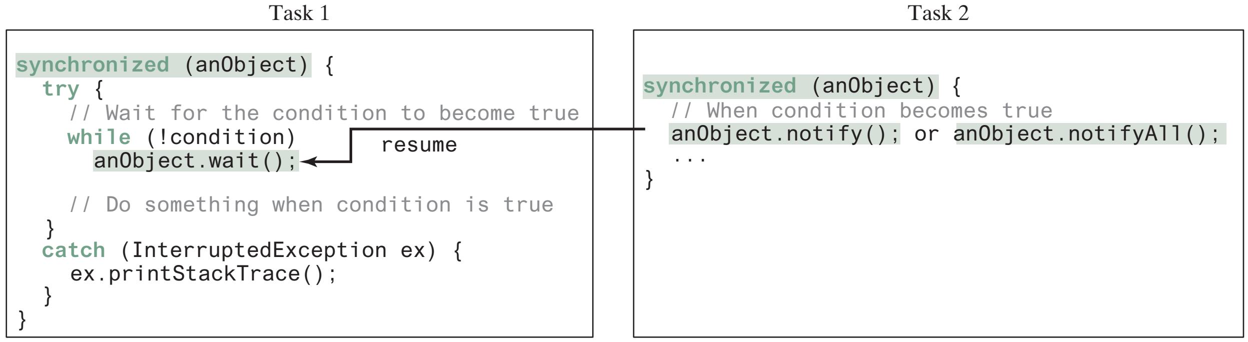
A condition is created from a `Lock` object. To invoke its method (e.g., `await()`, `signal()`, and `signalAll()`), you must first own the lock. If you invoke these methods without acquiring the lock, an `IllegalMonitorStateException` will be thrown.

Java's built-in monitor

Locks and conditions were introduced in Java 5. Prior to Java 5, thread communications were programmed using the object's built-in monitors. Locks and conditions are more powerful and flexible than the built-in monitor, so will not need to use monitors. However, if you are working with legacy Java code, you may encounter Java's built-in monitor.

monitor

A *monitor* is an object with mutual exclusion and synchronization capabilities. Only one thread can execute a method at a time in the monitor. A thread enters the monitor by acquiring a lock on it and exits by releasing the lock. *Any object can be a monitor*. An object becomes a monitor once a thread locks it. Locking is implemented using the `synchronized` keyword on a method or a block. A thread must acquire a lock before executing a synchronized method or block. A thread can wait in a monitor if the condition is not right for it to continue executing in the monitor. You can invoke the `wait()` method on the monitor object to release the lock so some other thread can get in the monitor and perhaps change the monitor's state. When the condition is right, the other thread can invoke the `notify()` or `notifyAll()` method to signal one or all waiting threads to regain the lock and resume execution. The template for invoking these methods is shown in Figure 32.17.



```
synchronized (anObject) {
  try {
    // Wait for the condition to become true
    while (!condition)
      anObject.wait();

    // Do something when condition is true
  }
  catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
```

```
synchronized (anObject) {
  // When condition becomes true
  anObject.notify(); or anObject.notifyAll();
  ...
}
```

Figure 32.17 The `wait()`, `notify()`, and `notifyAll()` methods coordinate thread communication.

The `wait()`, `notify()`, and `notifyAll()` methods must be called in a synchronized method or a synchronized block on the receiving object of these methods. Otherwise, an `IllegalMonitorStateException` will occur.

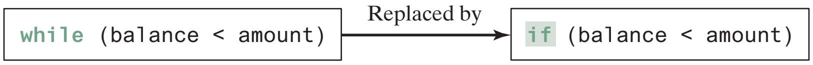
When `wait()` is invoked, it pauses the thread and simultaneously releases the lock on the object. When the thread is restarted after being notified, the lock is automatically reacquired.

The `wait()`, `notify()`, and `notifyAll()` methods on an object are analogous to the `await()`, `signal()`, and `signalAll()` methods on a condition.

Check Point

32.9.1 How do you create a condition on a lock? What are the `await()`, `signal()`, and `signalAll()` methods for?

32.9.2 What would happen if the `while` loop in line 58 of Listing 32.6 was changed to an `if` statement?



```
while (balance < amount)
```

```
if (balance < amount)
```

32.9.3 Why does the following class have a syntax error?

```
public class Test implements Runnable {
  public static void main(String[] args) {
    new Test();
  }

  public Test() throws InterruptedException {
    Thread thread = new Thread(this);
    thread.sleep(1000);
  }

  public synchronized void run() {
  }
}
```

32.9.4 What is a possible cause for `IllegalMonitorStateException`?

32.9.5 Can `wait()`, `notify()`, and `notifyAll()` be invoked from any object? What is the purpose of these methods?

32.9.6 What is wrong in the following code?

```
synchronized (object1) {
  try {
    while (!condition) object2.wait();
  }
  catch (InterruptedException ex) {
  }
}
```

32.10 Case Study: Producer/Consumer

Key Point

This section gives the classic Consumer/Producer example for demonstrating thread coordination.

Suppose that you use a buffer to store integers and that the buffer size is limited. The buffer provides the method `write(int)` to add an `int` value to the buffer and the method `read()` to read and delete an `int` value from the buffer. To synchronize the operations, use a lock with two conditions: `notEmpty` (i.e., the buffer is not empty) and `notFull` (i.e., the buffer is not full). When a task adds an `int` to the buffer, if the buffer is full, the task will wait for the `notFull` condition. When a task reads an `int` from the buffer, if the buffer is empty, the task will wait for the `notEmpty` condition. The interaction between the two tasks is shown in Figure 32.18.

Listing 32.7 presents the complete program. The program contains the `Buffer` class (lines 50–101) and two tasks for repeatedly adding and consuming numbers to and from the buffer (lines 16–47). The `write(int)` method (lines 62–79) adds an integer to the buffer. The `read()` method (lines 81–100) deletes and returns an integer from the buffer.

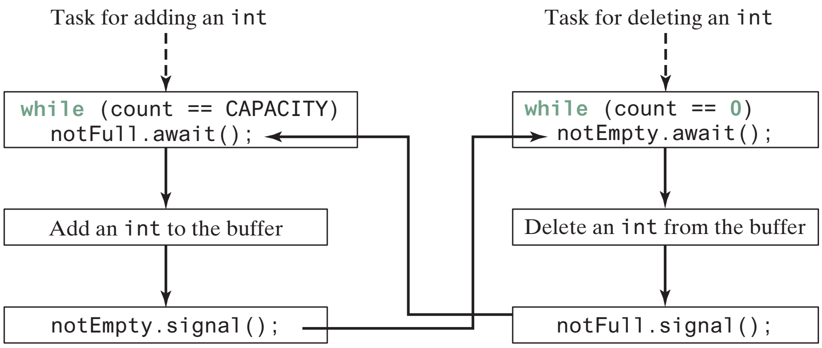


Figure 32.18 The conditions `notFull` and `notEmpty` are used to coordinate task interactions.

The buffer is actually a first-in, first-out queue (lines 52 and 53). The conditions `notEmpty` and `notFull` on the lock are created in lines 59 and 60. The conditions are bound to a lock. A lock must be acquired before a condition can be applied. If you use the `wait()` and `notify()` methods to rewrite this example, you have to designate two objects as monitors.

Listing 32.7 ConsumerProducer.java

```
 1  import java.util.concurrent.*;
 2  import java.util.concurrent.locks.*;
 3
 4  public class ConsumerProducer {
 5    private static Buffer buffer = new Buffer();
 6
 7    public static void main(String[] args) {
 8      // Create a thread pool with two threads
 9      ExecutorService executor = Executors.newFixedThreadPool(2);
10      executor.execute(new ProducerTask());
11      executor.execute(new ConsumerTask());
12      executor.shutdown();
13    }
14
15    // A task for adding an int to the buffer
16    private static class ProducerTask implements Runnable {
17      public void run() {
18        try {
19          int i = 1;
20          while (true) {
21            System.out.println("Producer writes " + i);
22            buffer.write(i++); // Add a value to the buffer
23            // Put the thread into sleep
24            Thread.sleep((int)(Math.random() * 10000));
25          }
26        }
27        catch (InterruptedException ex) {
28          ex.printStackTrace();
29        }
30      }
31    }
32
33    // A task for reading and deleting an int from the buffer
34    private static class ConsumerTask implements Runnable {
35      public void run() {
```

create a buffer

create two threads

producer task

consumer task

```
    try {
      while (true) {
        System.out.println("\t\t\tConsumer reads " + buffer.read());
        // Put the thread into sleep
        Thread.sleep((int)(Math.random() * 10000));
      }
    }
    catch (InterruptedException ex) {
      ex.printStackTrace();
    }
  }
}

// An inner class for buffer
private static class Buffer {
  private static final int CAPACITY = 1; // buffer size
  private java.util.LinkedList<Integer> queue =
    new java.util.LinkedList<>();

  // Create a new lock
  private static Lock lock = new ReentrantLock();

  // Create two conditions
  private static Condition notEmpty = lock.newCondition();
  private static Condition notFull = lock.newCondition();

  public void write(int value) {
    lock.lock(); // Acquire the lock
    try {
      while (queue.size() == CAPACITY) {
        System.out.println("Wait for notFull condition");
        notFull.await();
      }

      queue.offer(value);
      notEmpty.signal(); // Signal notEmpty condition
    }
    catch (InterruptedException ex) {
      ex.printStackTrace();
    }
    finally {
      lock.unlock(); // Release the lock
    }
  }

  public int read() {
    int value = 0;
    lock.lock(); // Acquire the lock
    try {
      while (queue.isEmpty()) {
        System.out.println("\t\t\tWait for notEmpty condition");
        notEmpty.await();
      }

      value = queue.remove();
      notFull.signal(); // Signal notFull condition
    }
    catch (InterruptedException ex) {
      ex.printStackTrace();
    }
```

create a lock

create a condition

create a condition

acquire the lock

wait for `notFull`

signal `notEmpty`

release the lock

acquire the lock

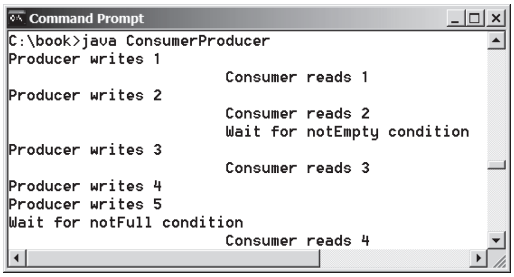
wait for `notEmpty`

signal `notFull`

release the lock

```
        finally {
          lock.unlock(); // Release the lock
          return value;
        }
      }
    }
  }
```

A sample run of the program is shown in Figure 32.19.



```
C:\book>java ConsumerProducer
Producer writes 1
                        Consumer reads 1
Producer writes 2
                        Consumer reads 2
                        Wait for notEmpty condition
Producer writes 3
                        Consumer reads 3
Producer writes 4
Producer writes 5
Wait for notFull condition
                        Consumer reads 4
```

FIGURE 32.19 Locks and conditions are used for communications between the Producer and Consumer threads.

Check Point

32.10.1 Can the `read` and `write` methods in the `Buffer` class be executed concurrently?

32.10.2 When invoking the `read` method, what happens if the queue is empty?

32.10.3 When invoking the `write` method, what happens if the queue is full?

Key Point

32.11 Blocking Queues

Java Collections Framework provides `ArrayBlockingQueue`, `LinkedBlockingQueue`, *and* `PriorityBlockingQueue` *for supporting blocking queues.*

blocking queue

Queues and priority queues were introduced in Section 20.9. A *blocking queue* causes a thread to block when you try to add an element to a full queue or to remove an element from an empty queue. The `BlockingQueue` interface extends `java.util.Queue` and provides the synchronized `put` and `take` methods for adding an element to the tail of the queue and for removing an element from the head of the queue, as shown in Figure 32.20.

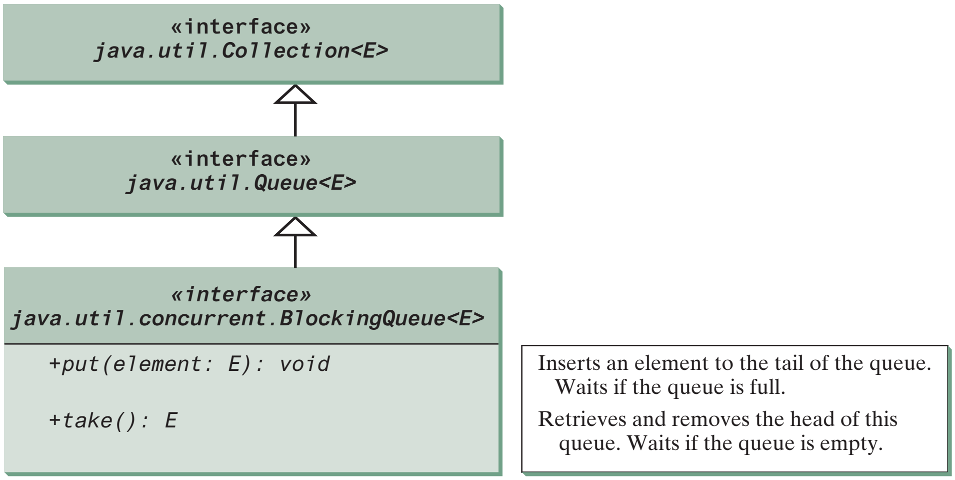

«interface»
java.util.Collection<E>
«interface»
java.util.Queue<E>
«interface»
java.util.concurrent.BlockingQueue<E>
+put(element: E): void
+take(): E
Inserts an element to the tail of the queue. Waits if the queue is full.
Retrieves and removes the head of this queue. Waits if the queue is empty.

FIGURE 32.20 `BlockingQueue` is a subinterface of `Queue`.

Three concrete blocking queues—ArrayBlockingQueue, LinkedBlockingQueue, and PriorityBlockingQueue—are provided in Java, as shown in Figure 32.21. All are in the java.util.concurrent package. ArrayBlockingQueue implements a blocking queue using an array. You have to specify a capacity or an optional fairness to construct an ArrayBlockingQueue. LinkedBlockingQueue implements a blocking queue using a linked list. You can create an unbounded or bounded LinkedBlockingQueue. PriorityBlockingQueue is a priority queue. You can create an unbounded or bounded priority queue.

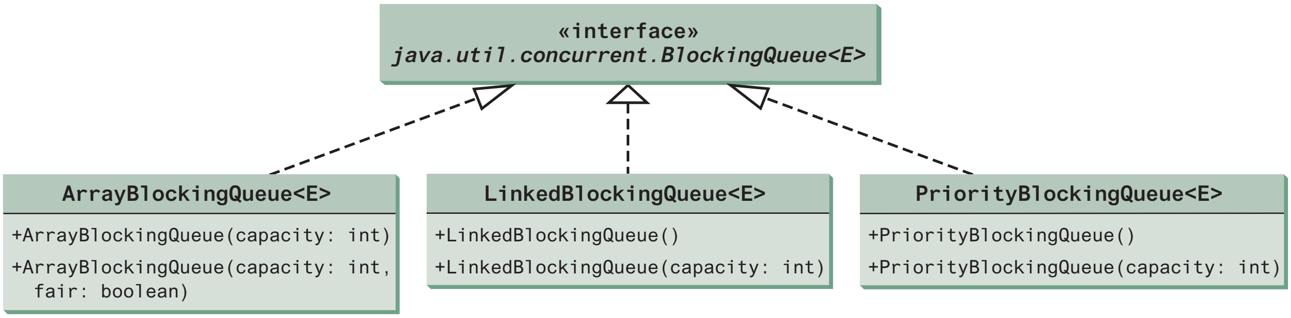

«interface»
java.util.concurrent.BlockingQueue<E>
ArrayBlockingQueue<E>
+ArrayBlockingQueue(capacity: int)
+ArrayBlockingQueue(capacity: int, fair: boolean)
LinkedBlockingQueue<E>
+LinkedBlockingQueue()
+LinkedBlockingQueue(capacity: int)
PriorityBlockingQueue<E>
+PriorityBlockingQueue()
+PriorityBlockingQueue(capacity: int)

FIGURE 32.21 ArrayBlockingQueue, LinkedBlockingQueue, and PriorityBlockingQueue are concrete blocking queues.

Note

The put method will never block an unbounded LinkedBlockingQueue or PriorityBlockingQueue.

unbounded queue

Listing 32.8 gives an example of using an ArrayBlockingQueue to simplify the Consumer/Producer example in Listing 32.10. Line 5 creates an ArrayBlockingQueue to store integers. The Producer thread puts an integer into the queue (line 22) and the Consumer thread takes an integer from the queue (line 38).

LISTING 32.8 ConsumerProducerUsingBlockingQueue.java

```
 1  import java.util.concurrent.*;
 2
 3  public class ConsumerProducerUsingBlockingQueue {
 4    private static ArrayBlockingQueue<Integer> buffer =
 5      new ArrayBlockingQueue<>(2);                                 create a buffer
 6
 7    public static void main(String[] args) {
 8      // Create a thread pool with two threads
 9      ExecutorService executor = Executors.newFixedThreadPool(2);  create two threads
10      executor.execute(new ProducerTask());
11      executor.execute(new ConsumerTask());
12      executor.shutdown();
13    }
14
15    // A task for adding an int to the buffer
16    private static class ProducerTask implements Runnable {       producer task
17      public void run() {
18        try {
19          int i = 1;
20          while (true) {
21            System.out.println("Producer writes " + i);
22            buffer.put(i++); // Add any value to the buffer, say, 1   put
23            // Put the thread into sleep
```

Consumer task

take

```
          Thread.sleep((int)(Math.random() * 10000));
        }
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }

  // A task for reading and deleting an int from the buffer
  private static class ConsumerTask implements Runnable {
    public void run() {
      try {
        while (true) {
          System.out.println("\t\t\tConsumer reads " + buffer.take());
          // Put the thread into sleep
          Thread.sleep((int)(Math.random() * 10000));
        }
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

In Listing 32.7, you used locks and conditions to synchronize the Producer and Consumer threads. This program does not use locks and conditions because synchronization is already implemented in `ArrayBlockingQueue`.

32.11.1 What is a blocking queue? What blocking queues are supported in Java?

32.11.2 What method do you use to add an element to an `ArrayBlockingQueue`? What happens if the queue is full?

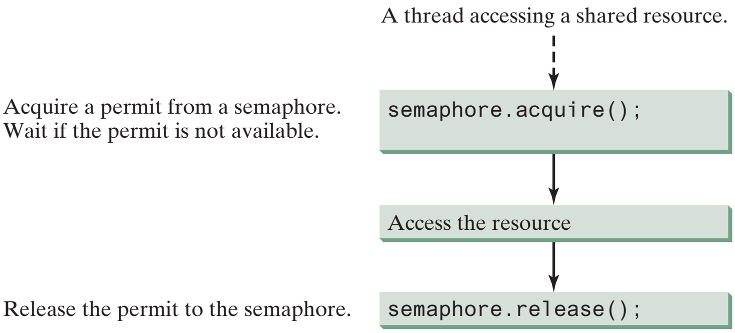
32.11.3 What method do you use to retrieve an element from an `ArrayBlockingQueue`? What happens if the queue is empty?

32.12 Semaphores

Semaphores can be used to restrict the number of threads that access a shared resource.

semaphore

In computer science, a *semaphore* is an object that controls the access to a common resource. Before accessing the resource, a thread must acquire a permit from the semaphore. After finishing with the resource, the thread must return the permit back to the semaphore, as shown in Figure 32.22.

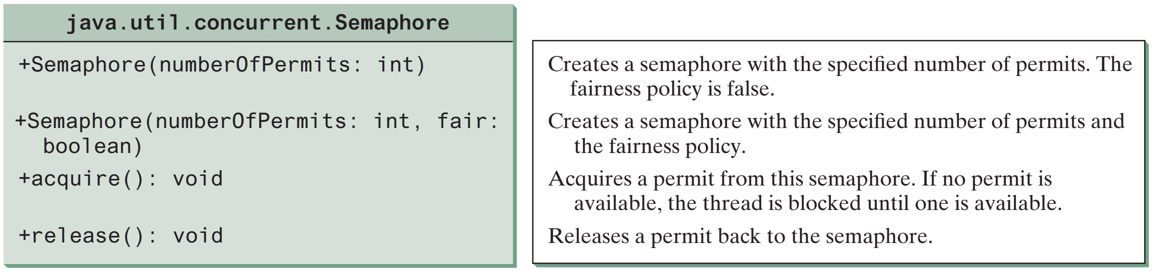

A thread accessing a shared resource.
Acquire a permit from a semaphore.
Wait if the permit is not available.
Access the resource
Release the permit to the semaphore.

```
semaphore.acquire();
```

```
semaphore.release();
```

FIGURE 32.22 A limited number of threads can access a shared resource controlled by a semaphore.

To create a semaphore, you have to specify the number of permits with an optional fairness policy, as shown in Figure 32.23. A task acquires a permit by invoking the semaphore's `acquire()` method and releases the permit by invoking the semaphore's `release()` method. Once a permit is acquired, the total number of available permits in a semaphore is reduced by `1`. Once a permit is released, the total number of available permits in a semaphore is increased by `1`.



java.util.concurrent.Semaphore

+Semaphore(numberOfPermits: int) — Creates a semaphore with the specified number of permits. The fairness policy is false.

+Semaphore(numberOfPermits: int, fair: boolean) — Creates a semaphore with the specified number of permits and the fairness policy.

+acquire(): void — Acquires a permit from this semaphore. If no permit is available, the thread is blocked until one is available.

+release(): void — Releases a permit back to the semaphore.

FIGURE 32.23 The `Semaphore` class contains the methods for accessing a semaphore.

A semaphore with just one permit can be used to simulate a mutually exclusive lock. Listing 32.9 revises the `Account` inner class in Listing 32.9 using a semaphore to ensure that only one thread at a time can access the `deposit` method.

LISTING 32.9 New Account Inner Class

```
 1  // An inner class for Account
 2  private static class Account {
 3    // Create a semaphore
 4    private static Semaphore semaphore = new Semaphore(1);
 5    private int balance = 0;
 6
 7    public int getBalance() {
 8      return balance;
 9    }
10
11    public void deposit(int amount) {
12      try {
13        semaphore.acquire(); // Acquire a permit
14        int newBalance = balance + amount;
15
16        // This delay is deliberately added to magnify the
17        // data-corruption problem and make it easy to see
18        Thread.sleep(5);
19
20        balance = newBalance;
21      }
22      catch (InterruptedException ex) {
23      }
24      finally {
25        semaphore.release(); // Release a permit
26      }
27    }
28  }
```

create a semaphore

acquire a permit

release a permit

A semaphore with one permit is created in line 4. A thread first acquires a permit when executing the deposit method in line 13. After the balance is updated, the thread releases the permit in line 25. It is a good practice to always place the `release()` method in the `finally` clause to ensure that the permit is finally released even in the case of exceptions.

32.12.1 What are the similarities and differences between a lock and a semaphore?

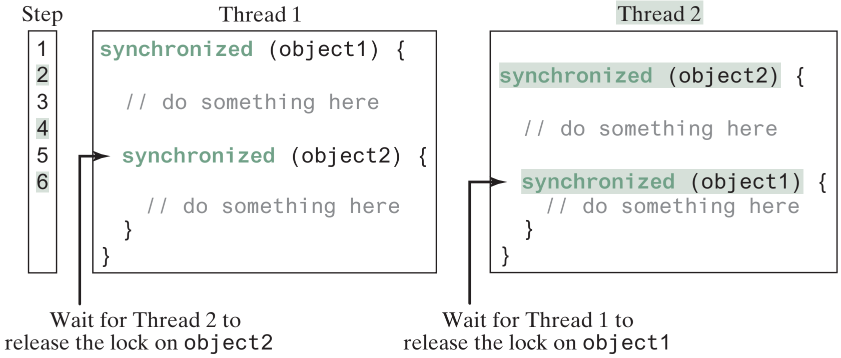
32.12.2 How do you create a semaphore that allows three concurrent threads? How do you acquire a semaphore? How do you release a semaphore?

32.13 Avoiding Deadlocks

Deadlocks can be avoided by using a proper resource ordering.

deadlock

Sometimes two or more threads need to acquire the locks on several shared objects. This could cause a *deadlock*, in which each thread has the lock on one of the objects and is waiting for the lock on the other object. Consider the scenario with two threads and two objects, as shown in Figure 32.24. Thread 1 has acquired a lock on `object1`, and Thread 2 has acquired a lock on `object2`. Now Thread 1 is waiting for the lock on `object2`, and Thread 2 for the lock on `object1`. Each thread waits for the other to release the lock it needs and until that happens, neither can continue to run.


Step
1
2
3
4
5
6
Thread 1
Thread 2
Wait for Thread 2 to release the lock on object2
Wait for Thread 1 to release the lock on object1

```
synchronized (object1) {

  // do something here

  synchronized (object2) {

    // do something here
  }
}
```

```
synchronized (object2) {

  // do something here

  synchronized (object1) {
    // do something here
  }
}
```

FIGURE 32.24 Thread 1 and Thread 2 are deadlocked.

resource ordering

Deadlock is easily avoided by using a simple technique known as *resource ordering*. With this technique, you assign an order to all the objects whose locks must be acquired and ensure that each thread acquires the locks in that order. For example in Figure 32.24, suppose the objects are ordered as `object1` and `object2`. Using the resource-ordering technique, Thread 2 must acquire a lock on `object1` first, then on `object2`. Once Thread 1 acquires a lock on `object1`, Thread 2 has to wait for a lock on `object1`. Thus, Thread 1 will be able to acquire a lock on `object2` and no deadlock will occur.

32.13.1 What is a deadlock? How can you avoid deadlock?

32.14 Thread States

A thread state indicates the status of thread.

Tasks are executed in threads. Threads can be in one of the five states: New, Ready, Running, Blocked, or Finished (see Figure 32.25).

When a thread is newly created, it enters the ***New*** state. After a thread is started by calling its `start()` method, it enters the ***Ready*** state. A ready thread is runnable but may not be running yet. The operating system has to allocate CPU time to it.

When a ready thread begins executing, it enters the ***Running*** state. A running thread can enter the ***Ready*** state if its given CPU time expires or its `yield()` method is called.

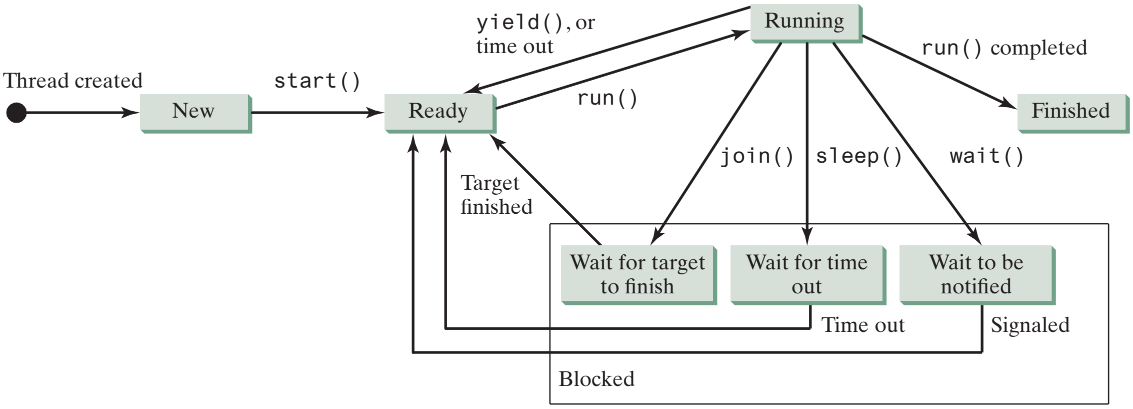


Figure 32.25 A thread can be in one of the five states: New, Ready, Running, Blocked, or Finished.

A thread can enter the ***Blocked*** state (i.e., become inactive) for several reasons. It may have invoked the `join()`, `sleep()`, or `wait()` method. It may be waiting for an I/O operation to finish. A blocked thread may be reactivated when the action inactivating it is reversed. For example, if a thread has been put to sleep and the sleep time has expired, the thread is reactivated and enters the **Ready** state.

Finally, a thread is ***Finished*** if it completes the execution of its `run()` method.

The `isAlive()` method is used to find out the state of a thread. It returns `true` if a thread is in the **Ready**, **Blocked**, or **Running** state; it returns `false` if a thread is new and has not started or if it is finished.

The `interrupt()` method interrupts a thread in the following way: If a thread is currently in the **Ready** or **Running** state, its interrupted flag is set; if a thread is currently blocked, it is awakened and enters the **Ready** state, and a `java.lang.InterruptedException` is thrown.

32.14.1 What is a thread state? Describe the states for a thread.

Check Point

32.15 Synchronized Collections

Key Point

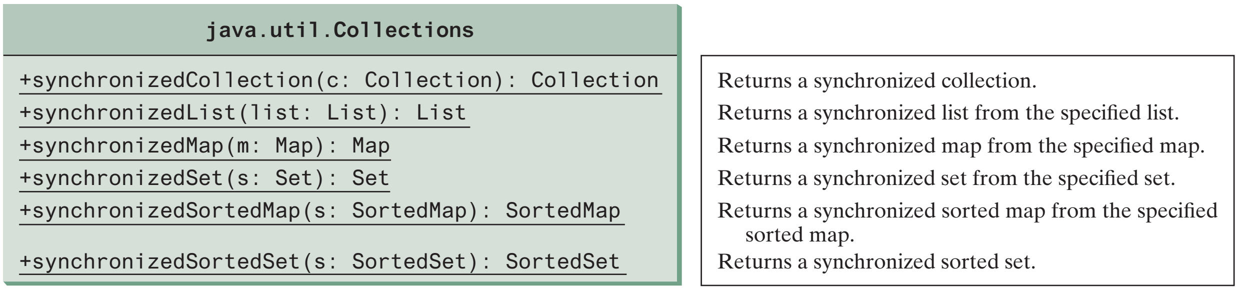
Java Collections Framework provides synchronized collections for lists, sets, and maps.

The classes in the Java Collections Framework are not thread-safe; that is, their contents may become corrupted if they are accessed and updated concurrently by multiple threads. You can protect the data in a collection by locking the collection or by using synchronized collections.

synchronized collection

The `Collections` class provides six static methods for wrapping a collection into a synchronized version, as shown in Figure 32.26. The collections created using these methods are called *synchronization wrappers*.

synchronization wrapper



java.util.Collections	
+synchronizedCollection(c: Collection): Collection	Returns a synchronized collection.
+synchronizedList(list: List): List	Returns a synchronized list from the specified list.
+synchronizedMap(m: Map): Map	Returns a synchronized map from the specified map.
+synchronizedSet(s: Set): Set	Returns a synchronized set from the specified set.
+synchronizedSortedMap(s: SortedMap): SortedMap	Returns a synchronized sorted map from the specified sorted map.
+synchronizedSortedSet(s: SortedSet): SortedSet	Returns a synchronized sorted set.

Figure 32.26 You can obtain synchronized collections using the methods in the `Collections` class.

Invoking synchronizedCollection(Collection c) returns a new Collection object, in which all the methods that access and update the original collection c are synchronized. These methods are implemented using the synchronized keyword. For example, the add method is implemented like this:

```
public boolean add(E o) {
  synchronized (this) {
    return c.add(o);
  }
}
```

Synchronized collections can be safely accessed and modified by multiple threads concurrently.

Note

The methods in java.util.Vector, java.util.Stack, and java.util.Hashtable are already synchronized. These are old classes introduced in JDK 1.0. Starting with JDK 1.5, you should use java.util.ArrayList to replace Vector, java.util.LinkedList to replace Stack, and java.util.Map to replace Hashtable. If synchronization is needed, use a synchronization wrapper.

fail-fast

The synchronization wrapper classes are thread-safe, but the iterator is *fail-fast*. This means that if you are using an iterator to traverse a collection while the underlying collection is being modified by another thread, then the iterator will immediately fail by throwing java.util.ConcurrentModificationException, which is a subclass of RuntimeException. To avoid this error, you need to create a synchronized collection object and acquire a lock on the object when traversing it. For example, to traverse a set, you have to write the code like this:

```
Set hashSet = Collections.synchronizedSet(new HashSet());

synchronized (hashSet) { // Must synchronize it
  Iterator iterator = hashSet.iterator();

  while (iterator.hasNext()) {
    System.out.println(iterator.next());
  }
}
```

Failure to do so may result in nondeterministic behavior, such as a ConcurrentModificationException.

Check Point

32.15.1 What is a synchronized collection? Is ArrayList synchronized? How do you make it synchronized?

32.15.2 Explain why an iterator is fail-fast.

Key Point

32.16 Parallel Programming

The Fork/Join Framework is used for parallel programming in Java.

Section 7.12 introduced the Arrays.sort and Arrays.parallelSort method for sorting an array. The parallelSort method utilizes multiple processors to reduce sort time. Chapter 22 introduced parallel streams for executing stream operations in parallel to speed up processing using multiple processors. The parallel processing are implemented using the Fork/Join Framework. This section, introduces the new Fork/Join Framework so you can write own code for parallel programming.

Fork/Join Framework

JDK 7 feature

The *Fork/Join Framework* is illustrated in Figure 32.27 (the diagram resembles a fork, hence its name). A problem is divided into nonoverlapping subproblems, which can be solved independently in parallel. The solutions to all subproblems are then joined to obtain an overall solution for the problem. This is the parallel implementation of the divide-and-conquer approach. In JDK 7's Fork/Join Framework, a *fork* can be viewed as an independent task that runs on a thread.

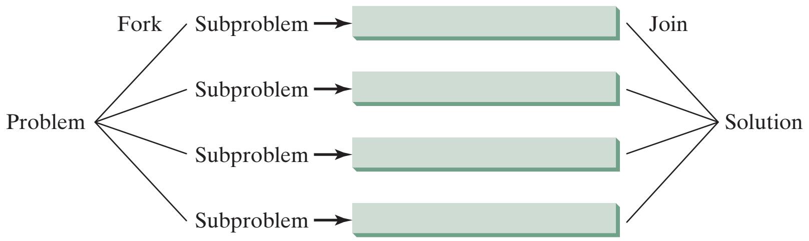


FIGURE 32.27 The nonoverlapping subproblems are solved in parallel.

The framework defines a task using the `ForkJoinTask` class, as shown in Figure 32.28 and executes a task in an instance of `ForkJoinPool`, as shown in Figure 32.29.

ForkJoinTask
ForkJoinPool

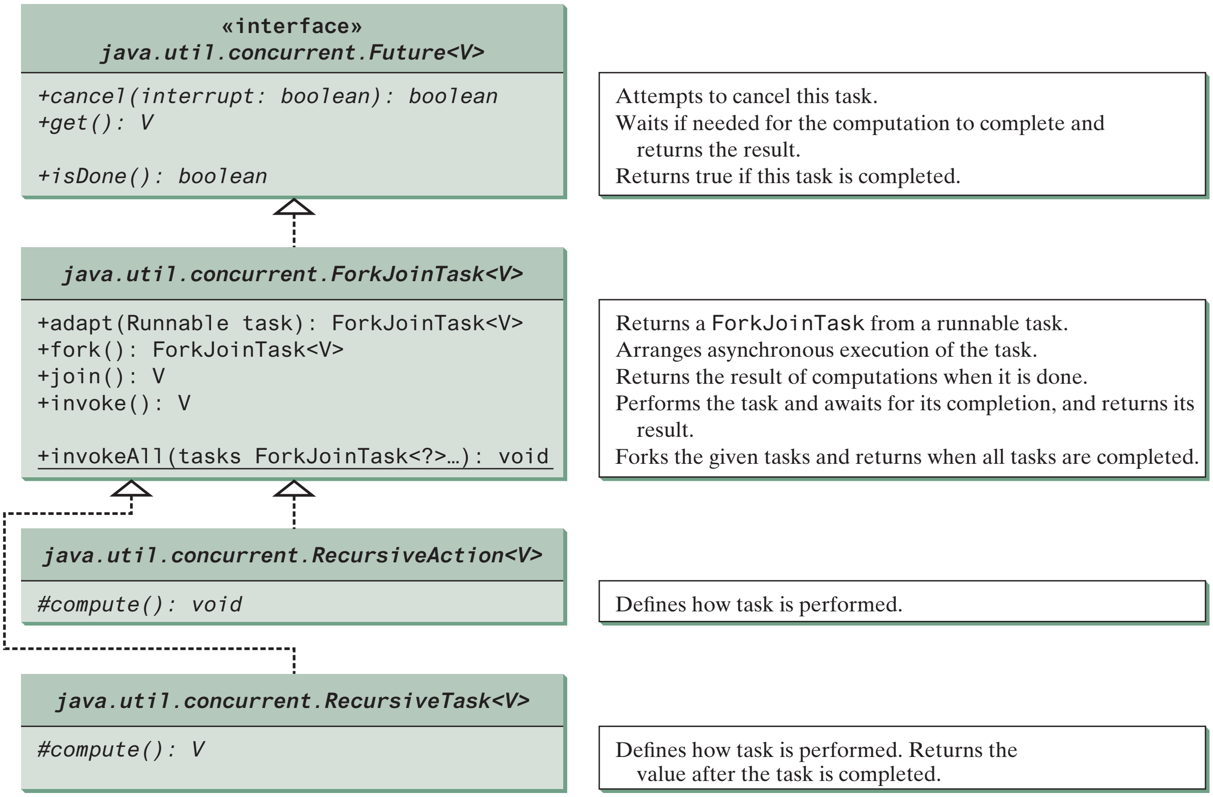

«interface»
java.util.concurrent.Future<V>
+cancel(interrupt: boolean): boolean
+get(): V
+isDone(): boolean
Attempts to cancel this task.
Waits if needed for the computation to complete and returns the result.
Returns true if this task is completed.
java.util.concurrent.ForkJoinTask<V>
+adapt(Runnable task): ForkJoinTask<V>
+fork(): ForkJoinTask<V>
+join(): V
+invoke(): V
+invokeAll(tasks ForkJoinTask<?>…): void
Returns a ForkJoinTask from a runnable task.
Arranges asynchronous execution of the task.
Returns the result of computations when it is done.
Performs the task and awaits for its completion, and returns its result.
Forks the given tasks and returns when all tasks are completed.
java.util.concurrent.RecursiveAction<V>
#compute(): void
Defines how task is performed.
java.util.concurrent.RecursiveTask<V>
#compute(): V
Defines how task is performed. Returns the value after the task is completed.

FIGURE 32.28 The `ForkJoinTask` class defines a task for asynchronous execution.

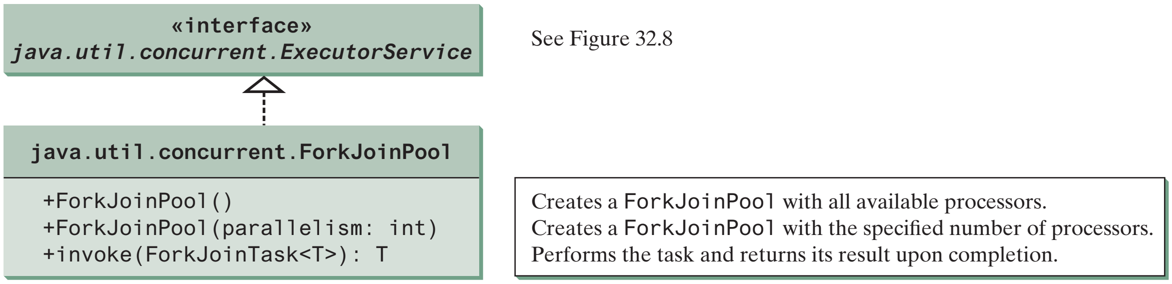

«interface»
java.util.concurrent.ExecutorService
See Figure 32.8
java.util.concurrent.ForkJoinPool
+ForkJoinPool()
+ForkJoinPool(parallelism: int)
+invoke(ForkJoinTask<T>): T
Creates a ForkJoinPool with all available processors.
Creates a ForkJoinPool with the specified number of processors.
Performs the task and returns its result upon completion.

FIGURE 32.29 The `ForkJoinPool` executes Fork/Join tasks.

ForkJoinTask is the abstract base class for tasks. A ForkJoinTask is a thread-like entity, but it is much lighter than a normal thread because huge numbers of tasks and subtasks can be executed by a small number of actual threads in a ForkJoinPool. The tasks are primarily coordinated using fork() and join(). Invoking fork() on a task arranges asynchronous execution and invoking join() waits until the task is completed. The invoke() and invokeAll(tasks) methods implicitly invoke fork() to execute the task and join() to wait for the tasks to complete and return the result, if any. Note the static method invokeAll takes a variable number of ForkJoinTask arguments using the ... syntax, which is introduced in Section 7.9.

The Fork/Join Framework is designed to parallelize divide-and-conquer solutions, which are naturally recursive. RecursiveAction and RecursiveTask are two subclasses of ForkJoinTask. To define a concrete task class, your class should extend RecursiveAction or RecursiveTask. RecursiveAction is for a task that doesn't return a value and RecursiveTask is for a task that does return a value. Your task class should override the compute() method to specify how a task is performed.

RecursiveAction

RecursiveTask

We now use a merge sort to demonstrate how to develop parallel programs using the Fork/Join Framework. The merge sort algorithm (introduced in Section 25.3) divides the array into two halves and applies a merge sort on each half recursively. After the two halves are sorted, the algorithm merges them. Listing 32.10 gives a parallel implementation of the merge sort algorithm and compares its execution time with a sequential sort.

Listing 32.10 ParallelMergeSort.java

```java
import java.util.concurrent.RecursiveAction;
import java.util.concurrent.ForkJoinPool;

public class ParallelMergeSort {
  public static void main(String[] args) {
    final int SIZE = 7000000;
    int[] list1 = new int[SIZE];
    int[] list2 = new int[SIZE];

    for (int i = 0; i < list1.length; i++)
      list1[i] = list2[i] = (int)(Math.random() * 10000000);

    long startTime = System.currentTimeMillis();
    parallelMergeSort(list1); // Invoke parallel merge sort
    long endTime = System.currentTimeMillis();
    System.out.println("\nParallel time with "
      + Runtime.getRuntime().availableProcessors() +
      " processors is " + (endTime - startTime) + " milliseconds");

    startTime = System.currentTimeMillis();
    MergeSort.mergeSort(list2); // MergeSort is in Listing 23.5
    endTime = System.currentTimeMillis();
    System.out.println("\nSequential time is " +
      (endTime - startTime) + " milliseconds");
  }

  public static void parallelMergeSort(int[] list) {
    RecursiveAction mainTask = new SortTask(list);
    ForkJoinPool pool = new ForkJoinPool();
    pool.invoke(mainTask);
  }

  private static class SortTask extends RecursiveAction {
    private final int THRESHOLD = 500;
```

invoke parallel sort

invoke sequential sort

create a ForkJoinTask

create a ForkJoinPool

execute a task

define concrete ForkJoinTask

```
35      private int[] list;
36
37      SortTask(int[] list) {
38        this.list = list;
39      }
40
41      @Override
42      protected void compute() {
43        if (list.length < THRESHOLD)
44          java.util.Arrays.sort(list);
45        else {
46          // Obtain the first half
47          int[] firstHalf = new int[list.length / 2];
48          System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
49
50          // Obtain the second half
51          int secondHalfLength = list.length - list.length / 2;
52          int[] secondHalf = new int[secondHalfLength];
53          System.arraycopy(list, list.length / 2,
54            secondHalf, 0, secondHalfLength);
55
56          // Recursively sort the two halves
57          invokeAll(new SortTask(firstHalf),
58            new SortTask(secondHalf));
59
60          // Merge firstHalf with secondHalf into list
61          MergeSort.merge(firstHalf, secondHalf, list);
62        }
63      }
64    }
65  }
```

perform the task

sort a small list

split into two parts

solve each part

merge two parts

```
Parallel time with two processors is 2829 milliseconds
Sequential time is 4751 milliseconds
```

Since the sort algorithm does not return a value, we define a concrete `ForkJoinTask` class by extending `RecursiveAction` (lines 33–64). The `compute` method is overridden to implement a recursive merge sort (lines 42–63). If the list is small, it is more efficient to be solved sequentially (line 44). For a large list, it is split into two halves (lines 47–54). The two halves are sorted concurrently (lines 57 and 58) and then merged (line 61).

The program creates a main `ForkJoinTask` (line 28), a `ForkJoinPool` (line 29), and places the main task for execution in a `ForkJoinPool` (line 30). The `invoke` method will return after the main task is completed.

When executing the main task, the task is split into subtasks, and the subtasks are invoked using the `invokeAll` method (lines 57 and 58). The `invokeAll` method will return after all the subtasks are completed. Note each subtask is further split into smaller tasks recursively. Huge numbers of subtasks may be created and executed in the pool. The Fork/Join Framework automatically executes and coordinates all the tasks efficiently.

The `MergeSort` class is defined in Listing 23.5. The program invokes `MergeSort.merge` to merge two sorted sublists (line 61). The program also invokes `MergeSort.mergeSort` (line 21) to sort a list using merge sort sequentially. You can see that the parallel sort is much faster than the sequential sort.

Note the loop for initializing the list can also be parallelized. However, you should avoid using `Math.random()` in the code because it is synchronized and cannot be executed in parallel (see Programming Exercise 32.12). The `parallelMergeSort` method only sorts an array of `int` values, but you can modify it to become a generic method (see Programming Exercise 32.13).

In general, a problem can be solved in parallel using the following pattern:

```
if (the program is small)
  solve it sequentially;
else {
  divide the problem into nonoverlapping subproblems;
  solve the subproblems concurrently;
  combine the results from subproblems to solve the whole problem;
}
```

Listing 32.11 develops a parallel method that finds the maximal number in a list.

LISTING 32.11 ParallelMax.java

```java
import java.util.concurrent.*;

public class ParallelMax {
  public static void main(String[] args) {
    // Create a list
    final int N = 9000000;
    int[] list = new int[N];
    for (int i = 0; i < list.length; i++)
      list[i] = i;

    long startTime = System.currentTimeMillis();
    System.out.println("\nThe maximal number is " + max(list));
    long endTime = System.currentTimeMillis();
    System.out.println("The number of processors is " +
      Runtime.getRuntime().availableProcessors());
    System.out.println("Time is " + (endTime - startTime)
      + " milliseconds");
  }

  public static int max(int[] list) {
    RecursiveTask<Integer> task = new MaxTask(list, 0, list.length);
    ForkJoinPool pool = new ForkJoinPool();
    return pool.invoke(task);
  }

  private static class MaxTask extends RecursiveTask<Integer> {
    private final static int THRESHOLD = 1000;
    private int[] list;
    private int low;
    private int high;

    public MaxTask(int[] list, int low, int high) {
      this.list = list;
      this.low = low;
      this.high = high;
    }

    @Override
    public Integer compute() {
      if (high - low < THRESHOLD) {
        int max = list[0];
        for (int i = low; i < high; i++)
          if (list[i] > max)
            max = list[i];
        return new Integer(max);
```

invoke `max`

create a `F°rkJoinTask`

create a `ForkJoinPool`

execute a task

define concrete `ForkJoinTask`

perform the task

solve a small problem

```
46       }
47       else {
48         int mid = (low + high) / 2;
49         RecursiveTask<Integer> left = new MaxTask(list, low, mid);
50         RecursiveTask<Integer> right = new MaxTask(list, mid, high);
51
52         right.fork();
53         left.fork();
54         return new Integer(Math.max(left.join().intValue(),
55           right.join().intValue()));
56       }
57     }
58   }
59 }
```

split into two parts

fork right

fork left

join tasks

```
The maximal number is 8999999
The number of processors is 2
Time is 44 milliseconds
```

Since the algorithm returns an integer, we define a task class for fork join by extending RecursiveTask<Integer> (lines 26–58). The compute method is overridden to return the max element in a list[low..high] (lines 39–57). If the list is small, it is more efficient to be solved sequentially (lines 40–46). For a large list, it is split into two halves (lines 48–50). The tasks left and right find the maximal element in the left half and right half, respectively. Invoking fork() on the task causes the task to be executed (lines 52 and 53). The join() method awaits for the task to complete and then returns the result (lines 54 and 55).

Check Point

32.16.1 How do you define a ForkJoinTask? What are the differences between RecursiveAction and RecursiveTask?

32.16.2 How do you tell the system to execute a task?

32.16.3 What method can you use to test if a task has been completed?

32.16.4 How do you create a ForkJoinPool? How do you place a task into a ForkJoinPool?

Key Terms

condition	32-18
deadlock	32-30
fail-fast	32-32
fairness policy	32-17
Fork/Join Framework	32-32
lock	32-16
monitor	32-22
multithreading	32-2
race condition	32-15
semaphore	32-28
synchronization wrapper	32-31
synchronized block	32-16
thread	32-2
thread-safe	32-15

Chapter Summary

1. Each task is an instance of the Runnable interface. A *thread* is an object that facilitates the execution of a task. You can define a task class by implementing the Runnable interface and create a thread by wrapping a task using a Thread constructor.

2. After a thread object is created, use the start() method to start a thread, and the sleep(long) method to put a thread to sleep so other threads get a chance to run.

3. A thread object never directly invokes the `run` method. The JVM invokes the `run` method when it is time to execute the thread. Your class must override the `run` method to tell the system what the thread will do when it runs.

4. To prevent threads from corrupting a shared resource, use *synchronized* methods or blocks. A *synchronized method* acquires a *lock* before it executes. In the case of an instance method, the lock is on the object for which the method was invoked. In the case of a static method, the lock is on the class.

5. A synchronized statement can be used to acquire a lock on any object, not just *this* object, when executing a block of the code in a method. This block is referred to as a *synchronized block*.

6. You can use explicit locks and *conditions* to facilitate communications among threads, as well as using the built-in monitor for objects.

7. The blocking queues (`ArrayBlockingQueue`, `LinkedBlockingQueue`, and `PriorityBlockingQueue`) provided in the Java Collections Framework provide automatical synchronization for the access to a queue.

8. You can use semaphores to restrict the number of concurrent accesses to a shared resource.

9. *Deadlock* occurs when two or more threads acquire locks on multiple objects and each has a lock on one object and is waiting for the lock on the other object. The *resource-ordering technique* can be used to avoid deadlock.

10. The JDK 7's Fork/Join Framework is designed for developing parallel programs. You can define a task class that extends `RecursiveAction` or `RecursiveTask` and execute the tasks concurrently in `ForkJoinPool` and obtain the overall solution after all tasks are completed.

Quiz

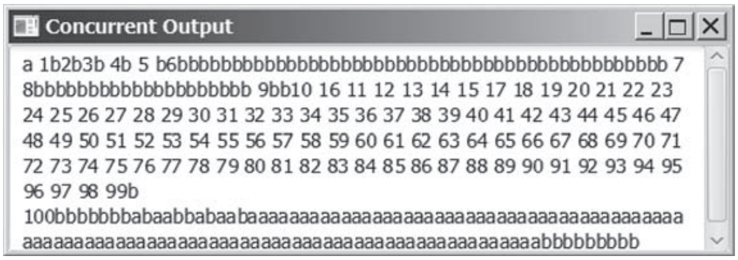
Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

Programming Exercises

Sections 32.1–32.5

***32.1** (*Revise Listing 32.1*) Rewrite Listing 32.1 to display the output in a text area, as shown in Figure 32.30.



```
a 1b2b3b 4b 5 b6bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb 7
8bbbbbbbbbbbbbbbbbb 9bb10 16 11 12 13 14 15 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
96 97 98 99b
100bbbbbbabaabbabaabaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbb
```

Figure 32.30 The output from three threads is displayed in a text area.

32.2 (*Racing cars*) Rewrite Programming Exercise 15.29 using a thread to control car racing. Compare the program with Programming Exercise 15.29 by setting the delay time to 10 in both the programs. Which one runs the animation faster?

32.3 (*Raise flags*) Rewrite Listing 15.13 using a thread to animate a flag being raised. Compare the program with Listing 15.13 by setting the delay time to 10 in both programs. Which one runs the animation faster?

Sections 32.8–32.12

32.4 (*Synchronize threads*) Write a program that launches 1,000 threads. Each thread adds `1` to a variable `sum` that initially is `0`. You need to pass `sum` by reference to each thread. In order to pass it by reference, define an `Integer` wrapper object to hold `sum`. Run the program with and without synchronization to see its effect.

32.5 (*Display a running fan*) Rewrite Programming Exercise 15.28 using a thread to control the fan animation.

32.6 (*Bouncing balls*) Rewrite Listing 15.17, BallPane.java using a thread to animate bouncing ball movements.

32.7 (*Control a clock*) Rewrite Programming Exercise 15.32 using a thread to control the clock animation.

32.8 (*Account synchronization*) Rewrite Listing 32.6, ThreadCooperation.java, using the object's `wait()` and `notifyAll()` methods.

32.9 (*Demonstrate* `ConcurrentModificationException`) The iterator is *fail-fast*. Write a program to demonstrate it by creating two threads that concurrently access and modify a set. The first thread creates a hash set filled with numbers and adds a new number to the set every second. The second thread obtains an iterator for the set and traverses the set back and forth through the iterator every second. You will receive a `ConcurrentModificationException` because the underlying set is being modified in the first thread while the set in the second thread is being traversed.

***32.10** (*Use synchronized sets*) Using synchronization, correct the problem in the preceding exercise so that the second thread does not throw a `ConcurrentModificationException`.

Section 32.15

***32.11** (*Demonstrate deadlock*) Write a program that demonstrates deadlock.

Section 32.18

***32.12** (*Parallel array initializer*) Implement the following method using the Fork/Join Framework to assign random values to the list.

```
public static void parallelAssignValues(double[] list)
```

Write a test program that creates a list with 9,000,000 elements and invokes `parallelAssignValues` to assign random values to the list. Also implement a sequential algorithm and compare the execution time of the two. Note if you use `Math.random()`, your parallel code execution time will be worse than the sequential code execution time because `Math.random()` is synchronized and cannot be executed in parallel. To fix this problem, create a `Random` object for assigning random values to a small list.

32.13 (*Generic parallel merge sort*) Revise Listing 32.10, ParallelMergeSort.java, to define a generic `parallelMergeSort` method as follows:

```
public static <E extends Comparable<E>> void
  parallelMergeSort(E[] list)
```

***32.14** (*Parallel quick sort*) Implement the following method in parallel to sort a list using quick sort (see Listing 23.7):

```
public static void parallelQuickSort(int[] list)
```

Write a test program that times the execution time for a list of size 9,000,000 using this parallel method and a sequential method.

***32.15** (*Parallel sum*) Implement the following method using Fork/Join to find the sum of a list.

```
public static double parallelSum(double[] list)
```

Write a test program that finds the sum in a list of 9,000,000 double values.

***32.16** (*Parallel matrix addition*) Programming Exercise 8.5 describes how to perform matrix addition. Suppose you have multiple processors, so you can speed up the matrix addition. Implement the following method in parallel:

```
public static double[][] parallelAddMatrix(
  double[][] a, double[][] b)
```

Write a test program that measures the execution time for adding two 2,000 × 2,000 matrices using the parallel method and sequential method, respectively.

***32.17** (*Parallel matrix multiplication*) Programming Exercise 7.6 describes how to perform matrix multiplication. Suppose that you have multiple processors, so you can speed up the matrix multiplication. Implement the following method in parallel:

```
public static double[][] parallelMultiplyMatrix(
  double[][] a, double[][] b)
```

Write a test program that measures the execution time for multiplying two 2,000 × 2,000 matrices using the parallel method and sequential method, respectively.

***32.18** (*Parallel Eight Queens*) Revise Listing 22.11, EightQueens.java, to develop a parallel algorithm that finds all solutions for the Eight Queens problem. (*Hint*: Launch eight subtasks, each of which places the queen in a different column in the first row.)

Comprehensive

*****32.19** (*Sorting animation*) Write an animation for selection sort, insertion sort, and bubble sort, as shown in Figure 32.31. Create an array of integers `1`, `2`, . . . , `50`. Shuffle it randomly. Create a pane to display the array in a histogram. You should invoke each sort method in a separate thread. Each algorithm uses two nested loops. When the algorithm completes an iteration in the outer loop, put the thread to sleep for 0.5 seconds and redisplay the array in the histogram. Color the last bar in the sorted subarray.

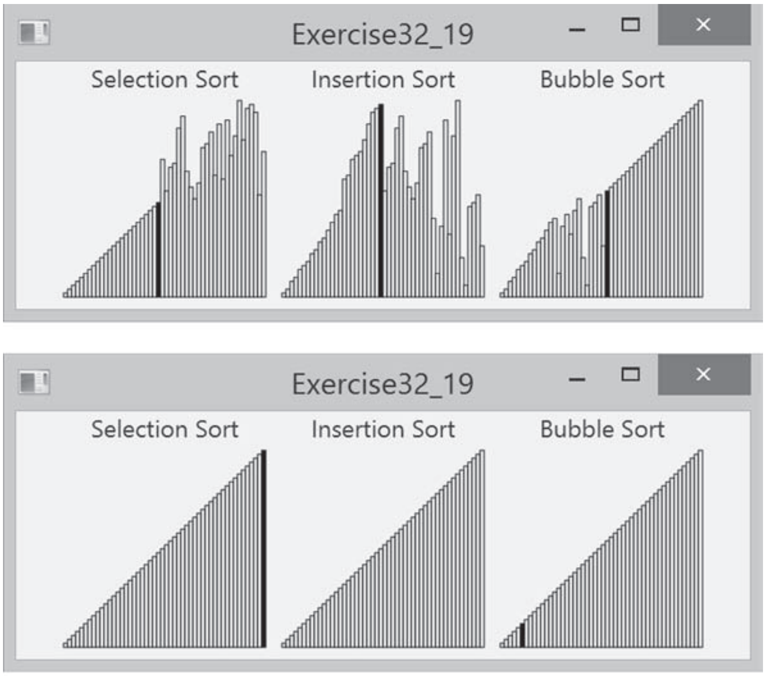
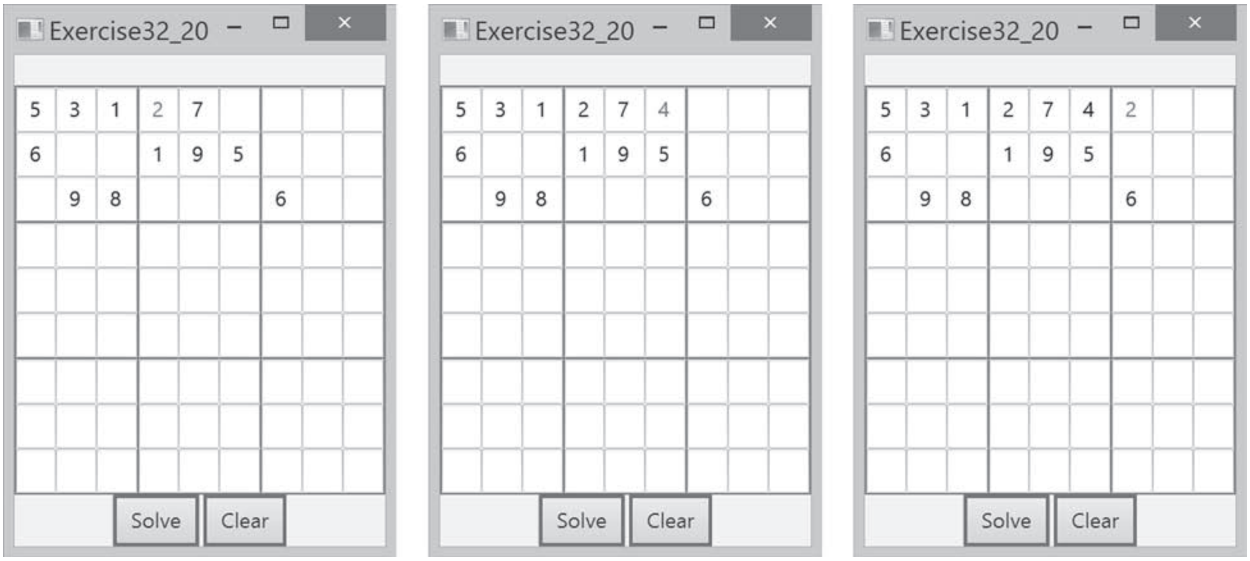


FIGURE 32.31 Three sorting algorithms are illustrated in the animation.

*****32.20** (*Sudoku search animation*) Modify Programming Exercise 22.21 to display the intermediate results of the search. Figure 32.32 gives a snapshot of an animation in progress with number 2 placed in the cell in Figure 32.32a, number 3 placed in the cell in Figure 32.32b, and number 3 placed in the cell in Figure 32.32c. The animation displays all the search steps.



5	3	1	2	7				
6			1	9	5			
	9	8				6		

(a)

5	3	1	2	7	4			
6			1	9	5			
	9	8				6		

(b)

5	3	1	2	7	4	2		
6			1	9	5			
	9	8				6		

(c)

FIGURE 32.32 The intermediate search steps are displayed in the animation for the Sudoku problem.

32.21 (*Combine colliding bouncing balls*) Rewrite Programming Exercise 20.5 using a thread to animate bouncing ball movements.

*****32.22** (*Eight Queens animation*) Modify Listing 22.11, EightQueens.java, to display the intermediate results of the search. As shown in Figure 32.33, the current row being searched is highlighted. Every one second, a new state of the chess board is displayed.

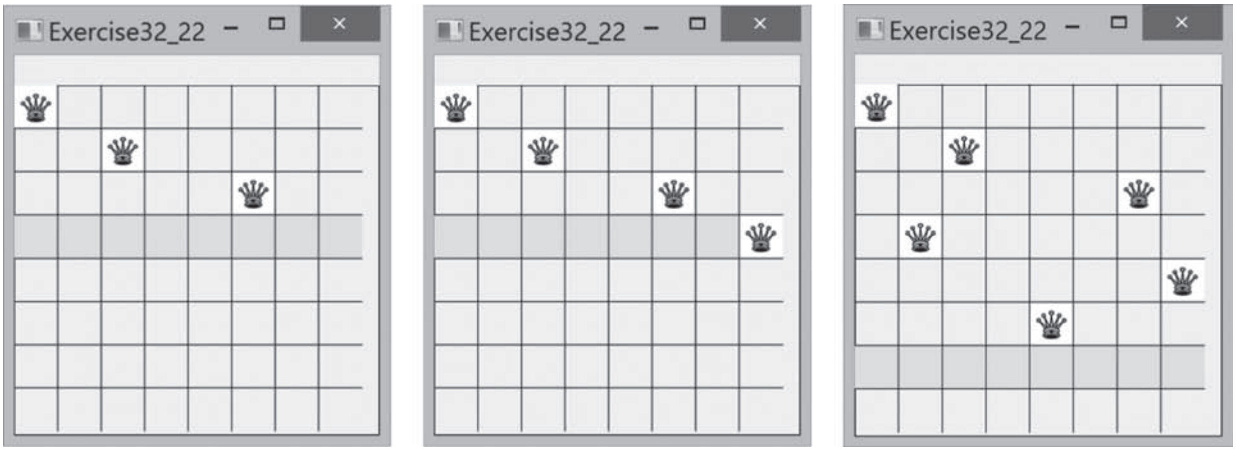


FIGURE 32.33 The intermediate search steps are displayed in the animation for the Eight Queens problem.

CHAPTER

33

Networking

Objectives

- To explain the terms: TCP, IP, domain name, domain name server, stream-based communications, and packet-based communications (§33.2).
- To create servers using server sockets (§33.2.1) and clients using client sockets (§33.2.2).
- To implement Java networking programs using stream sockets (§33.2.3).
- To develop an example of a client/server application (§33.2.4).
- To obtain Internet addresses using the `InetAddress` class (§33.3).
- To develop servers for multiple clients (§33.4).
- To send and receive objects on a network (§33.5).
- To develop an interactive tic-tac-toe game played on the Internet (§33.6).

33.1 Introduction

Computer networking is used to send and receive messages among computers on the Internet.

Key Point

To browse the Web or send an email, your computer must be connected to the Internet. The *Internet* is the global network of millions of computers. Your computer can connect to the Internet through an Internet Service Provider (ISP) using a dialup, DSL, or cable modem, or through a local area network (LAN).

IP address

When a computer needs to communicate with another computer, it needs to know the other computer's address. An *Internet Protocol* (IP) address uniquely identifies the computer on the Internet. An IP address consists of four dotted decimal numbers between **0** and **255**, such as **130.254.204.33.** Since it is not easy to remember so many numbers, they are often mapped to meaningful names called *domain names*, such as liang.armstrong.edu. Special servers called *Domain Name Servers* (DNS) on the Internet translate host names into IP addresses. When a computer contacts liang.armstrong.edu, it first asks the DNS to translate this domain name into a numeric IP address then sends the request using the IP address.

domain name
domain name server

TCP
UDP

The Internet Protocol is a low-level protocol for delivering data from one computer to another across the Internet in packets. Two higher-level protocols used in conjunction with the IP are the *Transmission Control Protocol* (TCP) and the *User Datagram Protocol* (UDP). TCP enables two hosts to establish a connection and exchange streams of data. TCP guarantees delivery of data and also guarantees that packets will be delivered in the same order in which they were sent. UDP is a standard, low-overhead, connectionless, host-to-host protocol that is used over the IP. UDP allows an application program on one computer to send a datagram to an application program on another computer.

stream-based communication
packet-based communication

Java supports both stream-based and packet-based communications. *Stream-based communications* use TCP for data transmission, whereas *packet-based communications* use UDP. Since TCP can detect lost transmissions and resubmit them, transmissions are lossless and reliable. UDP, in contrast, cannot guarantee lossless transmission. Stream-based communications are used in most areas of Java programming and are the focus of this chapter. Packet-based communications are introduced in Supplement III.P, Networking Using Datagram Protocol.

33.2 Client/Server Computing

Java provides the **ServerSocket** *class for creating a server socket, and the* **Socket** *class for creating a client socket. Two programs on the Internet communicate through a server socket and a client socket using I/O streams.*

Key Point

socket

Networking is tightly integrated in Java. The Java API provides the classes for creating sockets to facilitate program communications over the Internet. *Sockets* are the endpoints of logical connections between two hosts and can be used to send and receive data. Java treats socket communications much as it treats I/O operations; thus, programs can read from or write to sockets as easily as they can read from or write to files.

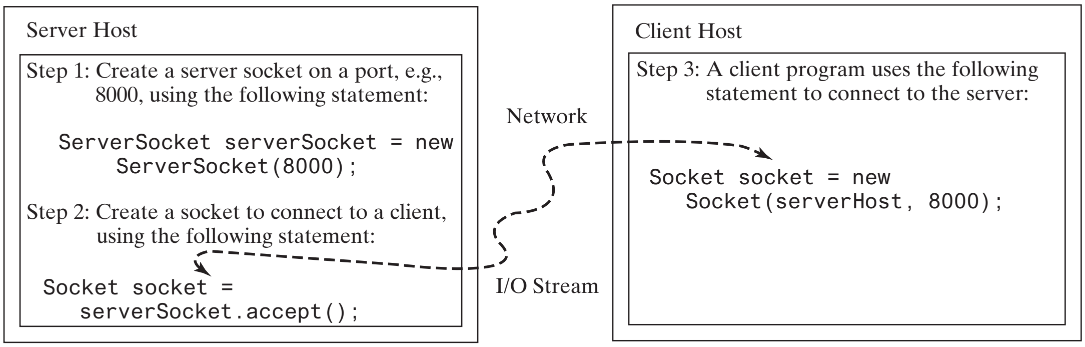
Network programming usually involves a server and one or more clients. The client sends requests to the server, and the server responds. The client begins by attempting to establish a connection to the server. The server can accept or deny the connection. Once a connection is established, the client and the server communicate through sockets.

The server must be running when a client attempts to connect to the server. The server waits for a connection request from the client. The statements needed to create sockets on a server and on a client are shown in Figure 33.1.

33.2.1 Server Sockets

server socket
port

To establish a server, you need to create a *server socket* and attach it to a *port*, which is where the server listens for connections. The port identifies the TCP service on the socket. Port numbers range from 0 to 65536, but port numbers 0 to 1024 are reserved for privileged services.


Server Host
Step 1: Create a server socket on a port, e.g., 8000, using the following statement:
Step 2: Create a socket to connect to a client, using the following statement:
Network
I/O Stream
Client Host
Step 3: A client program uses the following statement to connect to the server:

```
ServerSocket serverSocket = new
    ServerSocket(8000);
```

```
Socket socket =
    serverSocket.accept();
```

```
Socket socket = new
   Socket(serverHost, 8000);
```

FIGURE 33.1 The server creates a server socket and, once a connection to a client is established, connects to the client with a client socket.

For instance, the email server runs on port 25, and the Web server usually runs on port 80. You can choose any port number that is not currently used by other programs. The following statement creates a server socket `serverSocket`:

```
ServerSocket serverSocket = new ServerSocket(port);
```

> **Note**
> Attempting to create a server socket on a port already in use would cause a `java.net.BindException`.

`BindException`

33.2.2 Client Sockets

After a server socket is created, the server can use the following statement to listen for connections:

```
Socket socket = serverSocket.accept();
```

This statement waits until a client connects to the server socket. The client issues the following statement to request a connection to a server:

connect to client

```
Socket socket = new Socket(serverName, port);
```

This statement opens a socket so that the client program can communicate with the server. *`serverName`* is the server's Internet host name or IP address. The following statement creates a socket on the client machine to connect to the host 130.254.204.33 at port 8000:

client socket

use IP address

```
Socket socket = new Socket("130.254.204.33", 8000);
```

Alternatively, you can use the domain name to create a socket, as follows:

use domain name

```
Socket socket = new Socket("liang.armstrong.edu", 8000);
```

When you create a socket with a host name, the JVM asks the DNS to translate the host name into the IP address.

> **Note**
> A program can use the host name `localhost` or the IP address `127.0.0.1` to refer to the machine on which a client is running.

`localhost`

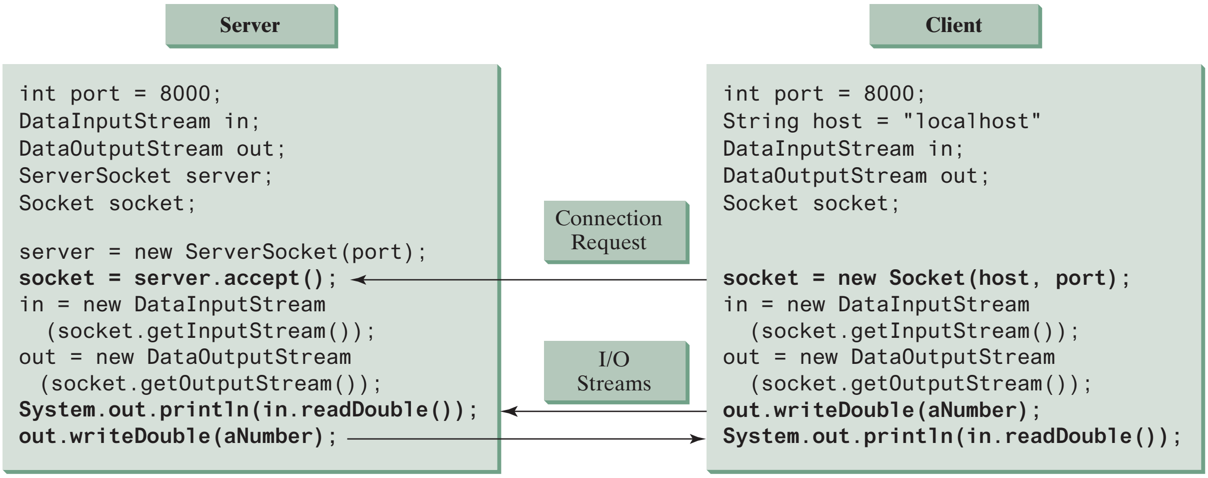
UnknownHostException

Note

The Socket constructor throws a java.net.UnknownHostException if the host cannot be found.

33.2.3 Data Transmission through Sockets

After the server accepts the connection, the communication between the server and the client is conducted in the same way as for I/O streams. The statements needed to create the streams and to exchange data between them are shown in Figure 33.2.



```
int port = 8000;
DataInputStream in;
DataOutputStream out;
ServerSocket server;
Socket socket;

server = new ServerSocket(port);
socket = server.accept();
in = new DataInputStream
  (socket.getInputStream());
out = new DataOutputStream
  (socket.getOutputStream());
System.out.println(in.readDouble());
out.writeDouble(aNumber);
```

```
int port = 8000;
String host = "localhost"
DataInputStream in;
DataOutputStream out;
Socket socket;

socket = new Socket(host, port);
in = new DataInputStream
  (socket.getInputStream());
out = new DataOutputStream
  (socket.getOutputStream());
out.writeDouble(aNumber);
System.out.println(in.readDouble());
```

FIGURE 33.2 The server and client exchange data through I/O streams on top of the socket.

To get an input stream and an output stream, use the getInputStream() and getOutputStream() methods on a socket object. For example, the following statements create an InputStream stream called input and an OutputStream stream called output from a socket:

```
InputStream input = socket.getInputStream();
OutputStream output = socket.getOutputStream();
```

The InputStream and OutputStream streams are used to read or write bytes. You can use DataInputStream, DataOutputStream, BufferedReader, and PrintWriter to wrap on the InputStream and OutputStream to read or write data, such as int, double, or String. The following statements, for instance, create the DataInputStream stream input and the DataOutputstream output to read and write primitive data values:

```
DataInputStream input = new DataInputStream
  (socket.getInputStream());
DataOutputStream output = new DataOutputStream
  (socket.getOutputStream());
```

The server can use input.readDouble() to receive a double value from the client, and output.writeDouble(d) to send the double value d to the client.

Tip

Recall that binary I/O is more efficient than text I/O because text I/O requires encoding and decoding. Therefore, it is better to use binary I/O for transmitting data between a server and a client to improve performance.

33.2.4 A Client/Server Example

This example presents a client program and a server program. The client sends data to a server. The server receives the data, uses it to produce a result, and then sends the result back to the client. The client displays the result on the console. In this example, the data sent from the client comprise the radius of a circle, and the result produced by the server is the area of the circle (see Figure 33.3).

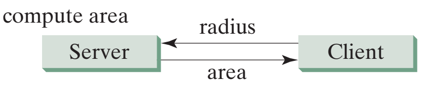


FIGURE 33.3 The client sends the radius to the server; the server computes the area and sends it to the client.

The client sends the radius through a `DataOutputStream` on the output stream socket, and the server receives the radius through the `DataInputStream` on the input stream socket, as shown in Figure 33.4a. The server computes the area and sends it to the client through a `DataOutputStream` on the output stream socket, and the client receives the area through a `DataInputStream` on the input stream socket, as shown in Figure 33.4b. The server and client programs are given in Listings 33.1 and 33.2. Figure 33.5 contains a sample run of the server and the client.

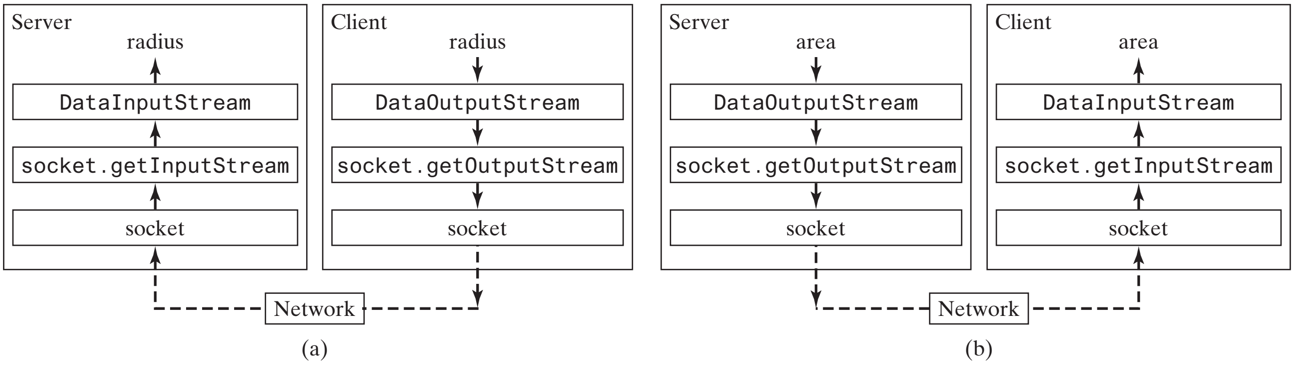
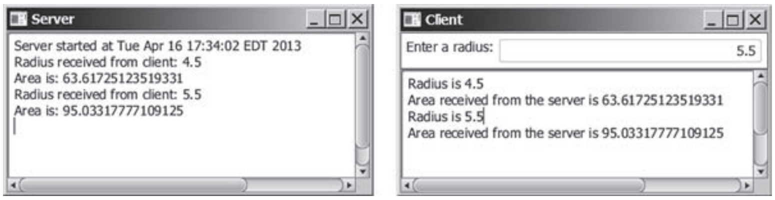


FIGURE 33.4 (a) The client sends the radius to the server. (b) The server sends the area to the client.



```
Server started at Tue Apr 16 17:34:02 EDT 2013
Radius received from client: 4.5
Area is: 63.61725123519331
Radius received from client: 5.5
Area is: 95.03317777109125
```

```
Radius is 4.5
Area received from the server is 63.61725123519331
Radius is 5.5
Area received from the server is 95.03317777109125
```

FIGURE 33.5 The client sends the radius to the server. The server receives it, computes the area, and sends the area to the client.

LISTING 33.1 Server.java

```java
import java.io.*;
import java.net.*;
import java.util.Date;
```

create server UI

server socket

update UI

connect client

input from client

output to client

read radius

write area

update UI

```
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.stage.Stage;

public class Server extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Text area for displaying contents
    TextArea ta = new TextArea();

    // Create a scene and place it in the stage
    Scene scene = new Scene(new ScrollPane(ta), 450, 200);
    primaryStage.setTitle("Server"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    new Thread(() -> {
      try {
        // Create a server socket
        ServerSocket serverSocket = new ServerSocket(8000);
        Platform.runLater(() ->
          ta.appendText("Server started at " + new Date() + '\n'));

        // Listen for a connection request
        Socket socket = serverSocket.accept();

        // Create data input and output streams
        DataInputStream inputFromClient = new DataInputStream(
          socket.getInputStream());
        DataOutputStream outputToClient = new DataOutputStream(
          socket.getOutputStream());

        while (true) {
          // Receive radius from the client
          double radius = inputFromClient.readDouble();

          // Compute area
          double area = radius * radius * Math.PI;

          // Send area back to the client
          outputToClient.writeDouble(area);

          Platform.runLater(() -> {
            ta.appendText("Radius received from client: "
              + radius + '\n');
            ta.appendText("Area is: " + area + '\n');
          });
        }
      }
    catch(IOException ex) {
      ex.printStackTrace();
    }
  }).start();
 }
}
```

Listing 33.2 Client.java

```java
import java.io.*;
import java.net.*;
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class Client extends Application {
  // IO streams
  DataOutputStream toServer = null;
  DataInputStream fromServer = null;

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Panel p to hold the label and text field
    BorderPane paneForTextField = new BorderPane();
    paneForTextField.setPadding(new Insets(5, 5, 5, 5));
    paneForTextField.setStyle("-fx-border-color: green");
    paneForTextField.setLeft(new Label("Enter a radius: "));

    TextField tf = new TextField();
    tf.setAlignment(Pos.BOTTOM_RIGHT);
    paneForTextField.setCenter(tf);

    BorderPane mainPane = new BorderPane();
    // Text area to display contents
    TextArea ta = new TextArea();
    mainPane.setCenter(new ScrollPane(ta));
    mainPane.setTop(paneForTextField);

    // Create a scene and place it in the stage
    Scene scene = new Scene(mainPane, 450, 200);
    primaryStage.setTitle("Client"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    tf.setOnAction(e -> {
      try {
        // Get the radius from the text field
        double radius = Double.parseDouble(tf.getText().trim());

        // Send the radius to the server
        toServer.writeDouble(radius);
        toServer.flush();

        // Get area from the server
        double area = fromServer.readDouble();

        // Display to the text area
        ta.appendText("Radius is " + radius + "\n");
        ta.appendText("Area received from the server is "
          + area + '\n');
```

create UI

handle action event

read radius

write radius

read area

request connection

input from server

output to server

```
59        }
60        catch (IOException ex) {
61          System.err.println(ex);
62        }
63      });
64
65      try {
66        // Create a socket to connect to the server
67        Socket socket = new Socket("localhost", 8000);
68        // Socket socket = new Socket("130.254.204.36", 8000);
69        // Socket socket = new Socket("drake.Armstrong.edu", 8000);
70
71        // Create an input stream to receive data from the server
72        fromServer = new DataInputStream(socket.getInputStream());
73
74        // Create an output stream to send data to the server
75        toServer = new DataOutputStream(socket.getOutputStream());
76      }
77      catch (IOException ex) {
78        ta.appendText(ex.toString() + '\n');
79      }
80    }
81  }
```

You start the server program first then start the client program. In the client program, enter a radius in the text field and press *Enter* to send the radius to the server. The server computes the area and sends it back to the client. This process is repeated until one of the two programs terminates.

The networking classes are in the package `java.net`. You should import this package when writing Java network programs.

The `Server` class creates a `ServerSocket serverSocket` and attaches it to port 8000 using this statement (line 26 in Server.java):

```
ServerSocket serverSocket = new ServerSocket(8000);
```

The server then starts to listen for connection requests, using the following statement (line 31 in Server.java):

```
Socket socket = serverSocket.accept();
```

The server waits until the client requests a connection. After it is connected, the server reads the radius from the client through an input stream, computes the area, and sends the result to the client through an output stream. The `ServerSocket accept()` method takes time to execute. It is not appropriate to run this method in the JavaFX application thread. So, we place it in a separate thread (lines 23–59). The statements for updating GUI needs to run from the JavaFX application thread using the `Platform.runLater` method (lines 27–28, 49–53).

The `Client` class uses the following statement to create a socket that will request a connection to the server on the same machine (localhost) at port 8000 (line 67 in Client.java).

```
Socket socket = new Socket("localhost", 8000);
```

If you run the server and the client on different machines, replace `localhost` with the server machine's host name or IP address. In this example, the server and the client are running on the same machine.

If the server is not running, the client program terminates with a `java.net.ConnectException`. After it is connected, the client gets input and output streams—wrapped by data input and output streams—in order to receive and send data to the server.

If you receive a `java.net.BindException` when you start the server, the server port is currently in use. You need to terminate the process that is using the server port then restart the server.

Note

When you create a server socket, you have to specify a port (e.g., 8000) for the socket. When a client connects to the server (line 67 in Client.java), a socket is created on the client. This socket has its own local port. This port number (e.g., 2047) is automatically chosen by the JVM, as shown in Figure 33.6.

client socket port

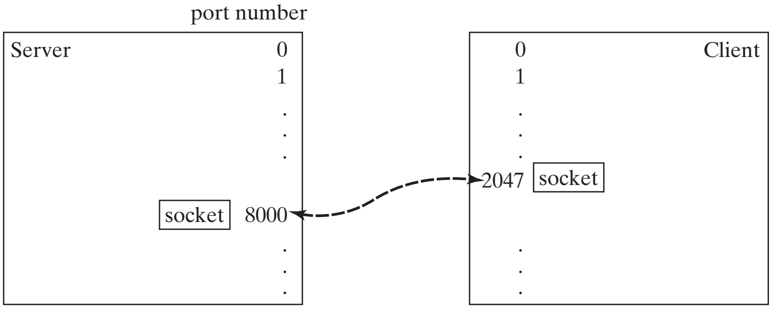


FIGURE 33.6 The JVM automatically chooses an available port to create a socket for the client.

To see the local port on the client, insert the following statement in line 70 in Client.java.

```
System.out.println("local port: " + socket.getLocalPort());
```

33.2.1 How do you create a server socket? What port numbers can be used? What happens if a requested port number is already in use? Can a port connect to multiple clients?

33.2.2 What are the differences between a server socket and a client socket?

33.2.3 How does a client program initiate a connection?

33.2.4 How does a server accept a connection?

33.2.5 How are data transferred between a client and a server?

Check Point

33.3 The InetAddress Class

The server program can use the **`InetAddress`** *class to obtain the information about the IP address and host name for the client.*

Key Point

Occasionally, you would like to know who is connecting to the server. You can use the `InetAddress` class to find the client's host name and IP address. The `InetAddress` class models an IP address. You can use the following statement in the server program to get an instance of `InetAddress` on a socket that connects to the client:

```
InetAddress inetAddress = socket.getInetAddress();
```

Next, you can display the client's host name and IP address, as follows:

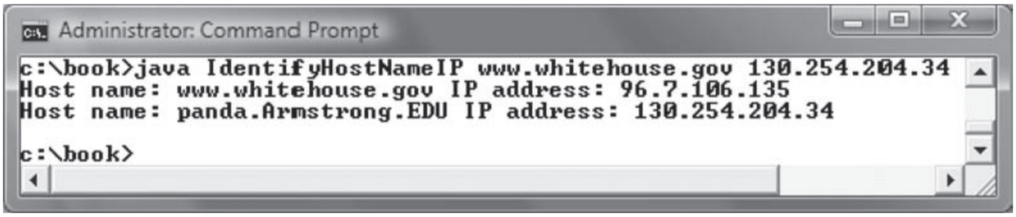
```
System.out.println("Client's host name is " +
  inetAddress.getHostName());
```

```
System.out.println("Client's IP Address is " +
  inetAddress.getHostAddress());
```

You can also create an instance of InetAddress from a host name or IP address using the static getByName method. For example, the following statement creates an InetAddress for the host liang.armstrong.edu.

```
InetAddress address = InetAddress.getByName("liang.armstrong.edu");
```

Listing 33.3 gives a program that identifies the host name and IP address of the arguments you pass in from the command line. Line 7 creates an InetAddress using the getByName method. Lines 8 and 9 use the getHostName and getHostAddress methods to get the host's name and IP address. Figure 33.7 shows a sample run of the program.



```
c:\book>java IdentifyHostNameIP www.whitehouse.gov 130.254.204.34
Host name: www.whitehouse.gov IP address: 96.7.106.135
Host name: panda.Armstrong.EDU IP address: 130.254.204.34

c:\book>
```

FIGURE 33.7 The program identifies host names and IP addresses.

LISTING 33.3 IdentifyHostNameIP.java

get an InetAddress
get host name
get host IP

```
 1  import java.net.*;
 2
 3  public class IdentifyHostNameIP {
 4    public static void main(String[] args) {
 5      for (int i = 0; i < args.length; i++) {
 6        try {
 7          InetAddress address = InetAddress.getByName(args[i]);
 8          System.out.print("Host name: " + address.getHostName() + " ");
 9          System.out.println("IP address: " + address.getHostAddress());
10        }
11        catch (UnknownHostException ex) {
12          System.err.println("Unknown host or IP address " + args[i]);
13        }
14      }
15    }
16  }
```

33.3.1 How do you obtain an instance of InetAddress?

33.3.2 What methods can you use to get the IP address and hostname from an InetAddress?

33.4 Serving Multiple Clients

A server can serve multiple clients. The connection to each client is handled by one thread.

Multiple clients are quite often connected to a single server at the same time. Typically, a server runs continuously on a server computer, and clients from all over the Internet can connect to it. You can use threads to handle the server's multiple clients simultaneously—simply create a thread for each connection. Here is how the server handles the establishment of a connection:

```
while (true) {
  Socket socket = serverSocket.accept(); // Connect to a client
  Thread thread = new ThreadClass(socket);
  thread.start();
}
```

The server socket can have many connections. Each iteration of the `while` loop creates a new connection. Whenever a connection is established, a new thread is created to handle communication between the server and the new client, and this allows multiple connections to run at the same time.

Listing 33.4 creates a server class that serves multiple clients simultaneously. For each connection, the server starts a new thread. This thread continuously receives input (the radius of a circle) from clients and sends the results (the area of the circle) back to them (see Figure 33.8). The client program is the same as in Listing 33.2. A sample run of the server with two clients is shown in Figure 33.9.

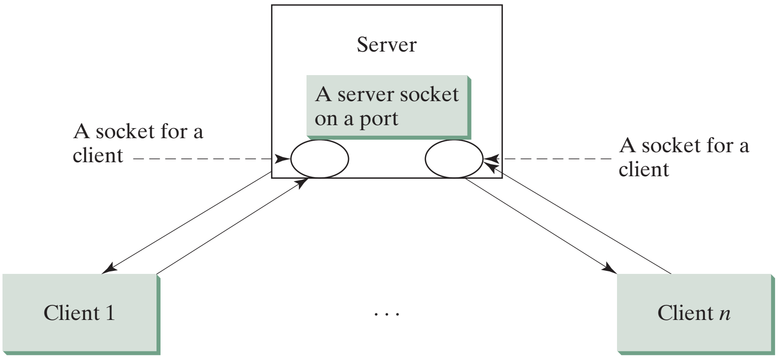
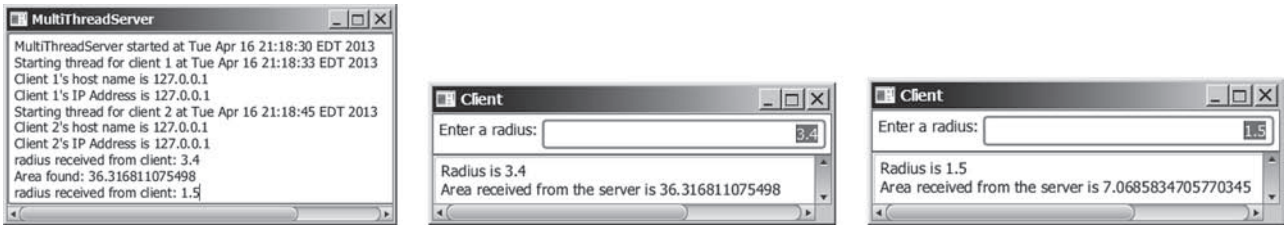


FIGURE 33.8 Multithreading enables a server to handle multiple independent clients.



```
MultiThreadServer started at Tue Apr 16 21:18:30 EDT 2013
Starting thread for client 1 at Tue Apr 16 21:18:33 EDT 2013
Client 1's host name is 127.0.0.1
Client 1's IP Address is 127.0.0.1
Starting thread for client 2 at Tue Apr 16 21:18:45 EDT 2013
Client 2's host name is 127.0.0.1
Client 2's IP Address is 127.0.0.1
radius received from client: 3.4
Area found: 36.316811075498
radius received from client: 1.5
```

Enter a radius: 3.4

```
Radius is 3.4
Area received from the server is 36.316811075498
```

Enter a radius: 1.5

```
Radius is 1.5
Area received from the server is 7.0685834705770345
```

FIGURE 33.9 The server spawns a thread in order to serve a client.

LISTING 33.4 MultiThreadServer.java

```
import java.io.*;
import java.net.*;
import java.util.Date;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.stage.Stage;

```

```
                        public class MultiThreadServer extends Application {
                          // Text area for displaying contents
                          private TextArea ta = new TextArea();

                          // Number a client
                          private int clientNo = 0;

                          @Override // Override the start method in the Application class
                          public void start(Stage primaryStage) {
                            // Create a scene and place it in the stage
                            Scene scene = new Scene(new ScrollPane(ta), 450, 200);
                            primaryStage.setTitle("MultiThreadServer"); // Set the stage title
                            primaryStage.setScene(scene); // Place the scene in the stage
                            primaryStage.show(); // Display the stage

                            new Thread( () -> {
                              try {
                                // Create a server socket
server socket                   ServerSocket serverSocket = new ServerSocket(8000);
                                ta.appendText("MultiThreadServer started at "
                                  + new Date() + '\n');

                                while (true) {
                                  // Listen for a new connection request
connect client                    Socket socket = serverSocket.accept();

                                  // Increment clientNo
                                  clientNo++;

update GUI                        Platform.runLater( () -> {
                                    // Display the client number
                                    ta.appendText("Starting thread for client " + clientNo +
                                      " at " + new Date() + '\n');

                                    // Find the client's host name, and IP address
network information                 InetAddress inetAddress = socket.getInetAddress();
                                    ta.appendText("Client " + clientNo + "'s host name is "
                                      + inetAddress.getHostName() + "\n");
                                    ta.appendText("Client " + clientNo + "'s IP Address is"
                                      + inetAddress.getHostAddress() + "\n");
                                  });

                                  // Create and start a new thread for the connection
create task                       new Thread(new HandleAClient(socket)).start();
                                }
                              }
                              catch(IOException ex) {
                                System.err.println(ex);
                              }
start thread                }).start();
                          }

                          // Define the thread class for handling new connection
                          class HandleAClient implements Runnable {
task class                  private Socket socket; // A connected socket

                            /** Construct a thread */
                            public HandleAClient(Socket socket) {
                              this.socket = socket;
                            }

```

```
 72      /** Run a thread */
 73      public void run() {
 74        try {
 75          // Create data input and output streams
 76          DataInputStream inputFromClient = new DataInputStream(
 77            socket.getInputStream());
 78          DataOutputStream outputToClient = new DataOutputStream(
 79            socket.getOutputStream());
 80
 81          // Continuously serve the client
 82          while (true) {
 83            // Receive radius from the client
 84            double radius = inputFromClient.readDouble();
 85
 86            // Compute area
 87            double area = radius * radius * Math.PI;
 88
 89            // Send area back to the client
 90            outputToClient.writeDouble(area);
 91
 92            Platform.runLater(() -> {
 93              ta.appendText("radius received from client: " +
 94                radius + '\n');
 95              ta.appendText("Area found: " + area + '\n');
 96            });
 97          }
 98        }
 99        catch(IOException ex) {
100          ex.printStackTrace();
101        }
102      }
103    }
104  }
```

I/O

update GUI

The server creates a server socket at port 8000 (line 29) and waits for a connection (line 35). When a connection with a client is established, the server creates a new thread to handle the communication (line 54). It then waits for another connection in an infinite `while` loop (lines 33–55).

The threads, which run independently of one another, communicate with designated clients. Each thread creates data input and output streams that receive and send data to a client.

33.4.1 How do you make a server serve multiple clients?

Check Point

33.5 Sending and Receiving Objects

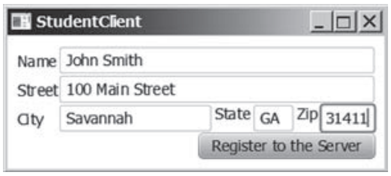
A program can send and receive objects from another program.

Key Point

In the preceding examples, you learned how to send and receive data of primitive types. You can also send and receive objects using `ObjectOutputStream` and `ObjectInputStream` on socket streams. To enable passing, the objects must be serializable. The following example demonstrates how to send and receive objects.

The example consists of three classes: StudentAddress.java (Listing 33.5), StudentClient.java (Listing 33.6), and StudentServer.java (Listing 33.7). The client program collects student information from the client and sends it to a server, as shown in Figure 33.10.

The `StudentAddress` class contains the student information: name, street, city, state, and zip. The `StudentAddress` class implements the `Serializable` interface. Therefore, a `StudentAddress` object can be sent and received using the object output and input streams.



Name John Smith
Street 100 Main Street
City Savannah State GA Zip 31411

FIGURE 33.10 The client sends the student information in an object to the server.

LISTING 33.5 StudentAddress.java

serialized

```java
public class StudentAddress implements java.io.Serializable {
  private String name;
  private String street;
  private String city;
  private String state;
  private String zip;

  public StudentAddress(String name, String street, String city,
    String state, String zip) {
    this.name = name;
    this.street = street;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  public String getName() {
    return name;
  }

  public String getStreet() {
    return street;
  }

  public String getCity() {
    return city;
  }

  public String getState() {
    return state;
  }

  public String getZip() {
    return zip;
  }
}
```

The client sends a `StudentAddress` object through an `ObjectOutputStream` on the output stream socket, and the server receives the `Student` object through the `ObjectInputStream` on the input stream socket, as shown in Figure 33.11. The client uses the `writeObject` method in the `ObjectOutputStream` class to send data about a student to the server, and the server receives the student's information using the `readObject` method in the `ObjectInputStream` class. The server and client programs are given in Listings 33.6 and 33.7.

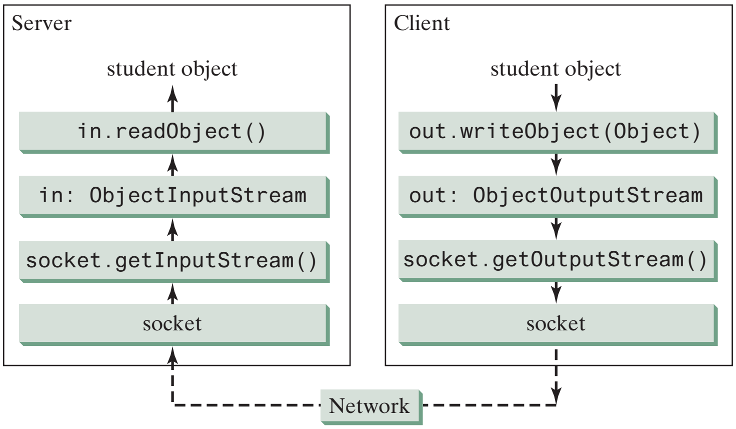


FIGURE 33.11 The client sends a `StudentAddress` object to the server.

LISTING 33.6 StudentClient.java

```java
import java.io.*;
import java.net.*;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.HPos;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class StudentClient extends Application {
  private TextField tfName = new TextField();
  private TextField tfStreet = new TextField();
  private TextField tfCity = new TextField();
  private TextField tfState = new TextField();
  private TextField tfZip = new TextField();

  // Button for sending a student to the server
  private Button btRegister = new Button("Register to the Server");

  // Host name or ip
  String host = "localhost";

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    GridPane pane = new GridPane();
    pane.add(new Label("Name"), 0, 0);
    pane.add(tfName, 1, 0);
    pane.add(new Label("Street"), 0, 1);
    pane.add(tfStreet, 1, 1);
    pane.add(new Label("City"), 0, 2);

```

create UI

register listener

server socket

output stream

send to server

```
    HBox hBox = new HBox(2);
    pane.add(hBox, 1, 2);
    hBox.getChildren().addAll(tfCity, new Label("State"), tfState,
      new Label("Zip"), tfZip);
    pane.add(btRegister, 1, 3);
    GridPane.setHalignment(btRegister, HPos.RIGHT);

    pane.setAlignment(Pos.CENTER);
    tfName.setPrefColumnCount(15);
    tfStreet.setPrefColumnCount(15);
    tfCity.setPrefColumnCount(10);
    tfState.setPrefColumnCount(2);
    tfZip.setPrefColumnCount(3);

    btRegister.setOnAction(new ButtonListener());

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane, 450, 200);
    primaryStage.setTitle("StudentClient"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  /** Handle button action */
  private class ButtonListener implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent e) {
      try {
        // Establish connection with the server
        Socket socket = new Socket(host, 8000);

        // Create an output stream to the server
        ObjectOutputStream toServer =
          new ObjectOutputStream(socket.getOutputStream());

        // Get text field
        String name = tfName.getText().trim();
        String street = tfStreet.getText().trim();
        String city = tfCity.getText().trim();
        String state = tfState.getText().trim();
        String zip = tfZip.getText().trim();

        // Create a Student object and send to the server
        StudentAddress s =
          new StudentAddress(name, street, city, state, zip);
        toServer.writeObject(s);
      }
      catch (IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

LISTING 33.7 StudentServer.java

```
import java.io.*;
import java.net.*;

public class StudentServer {
 5     private ObjectOutputStream outputToFile;
 6     private ObjectInputStream inputFromClient;
 7
 8     public static void main(String[] args) {
 9       new StudentServer();
10     }
11
12     public StudentServer() {
13       try {
14         // Create a server socket
15         ServerSocket serverSocket = new ServerSocket(8000);
16         System.out.println("Server started ");
17
18         // Create an object output stream
19         outputToFile = new ObjectOutputStream(
20           new FileOutputStream("student.dat", true));
21
22         while (true) {
23           // Listen for a new connection request
24           Socket socket = serverSocket.accept();
25
26           // Create an input stream from the socket
27           inputFromClient =
28             new ObjectInputStream(socket.getInputStream());
29
30           // Read from input
31           Object object = inputFromClient.readObject();
32
33           // Write to the file
34           outputToFile.writeObject(object);
35           System.out.println("A new student object is stored");
36         }
37       }
38       catch(ClassNotFoundException ex) {
39         ex.printStackTrace();
40       }
41       catch(IOException ex) {
42         ex.printStackTrace();
43       }
44       finally {
45         try {
46           inputFromClient.close();
47           outputToFile.close();
48         }
49         catch (Exception ex) {
50           ex.printStackTrace();
51         }
52       }
53     }
54   }
```

server socket

output to file

connect to client

input stream

get from client

write to file

On the client side, when the user clicks the *Register to the Server* button, the client creates a socket to connect to the host (line 67), creates an ObjectOutputStream on the output stream of the socket (lines 70 and 71), and invokes the writeObject method to send the StudentAddress object to the server through the object output stream (line 83).

On the server side, when a client connects to the server, the server creates an ObjectInputStream on the input stream of the socket (lines 27 and 28), invokes the readObject method to receive the StudentAddress object through the object input stream (line 31), and writes the object to a file (line 34).

Check Point

33.5.1 How does a server receive connection from a client? How does a client connect to a server?

33.5.2 How do you find the host name of a client program from the server?

33.5.3 How do you send and receive an object?

33.6 Case Study: Distributed Tic-Tac-Toe Games

Key Point

This section develops a program that enables two players to play the tic-tac-toe game on the Internet.

In Section 16.12, Case Study: Developing a Tic-Tac-Toe Game, you developed a program for a tic-tac-toe game that enables two players to play the game on the same machine. In this section, you will learn how to develop a distributed tic-tac-toe game using multithreads and networking with socket streams. A distributed tic-tac-toe game enables users to play on different machines from anywhere on the Internet.

You need to develop a server for multiple clients. The server creates a server socket and accepts connections from every two players to form a session. Each session is a thread that communicates with the two players and determines the status of the game. The server can establish any number of sessions, as shown in Figure 33.12.

For each session, the first client connecting to the server is identified as player 1 with token `X`, and the second client connecting is identified as player 2 with token `O`. The server notifies the players of their respective tokens. Once two clients are connected to it, the server starts a thread to facilitate the game between the two players by performing the steps repeatedly, as shown in Figure 33.13.

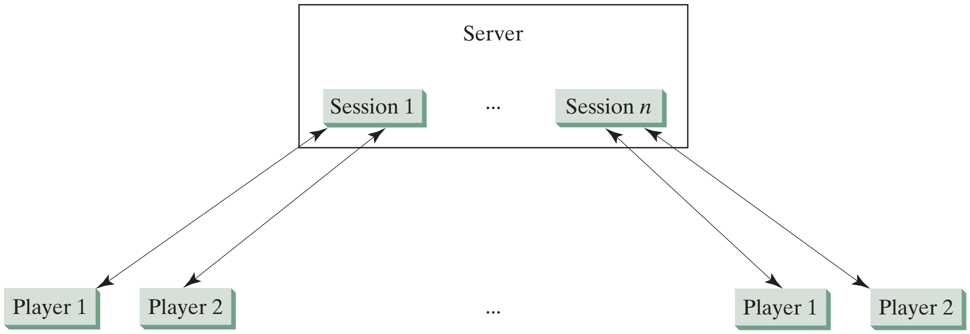


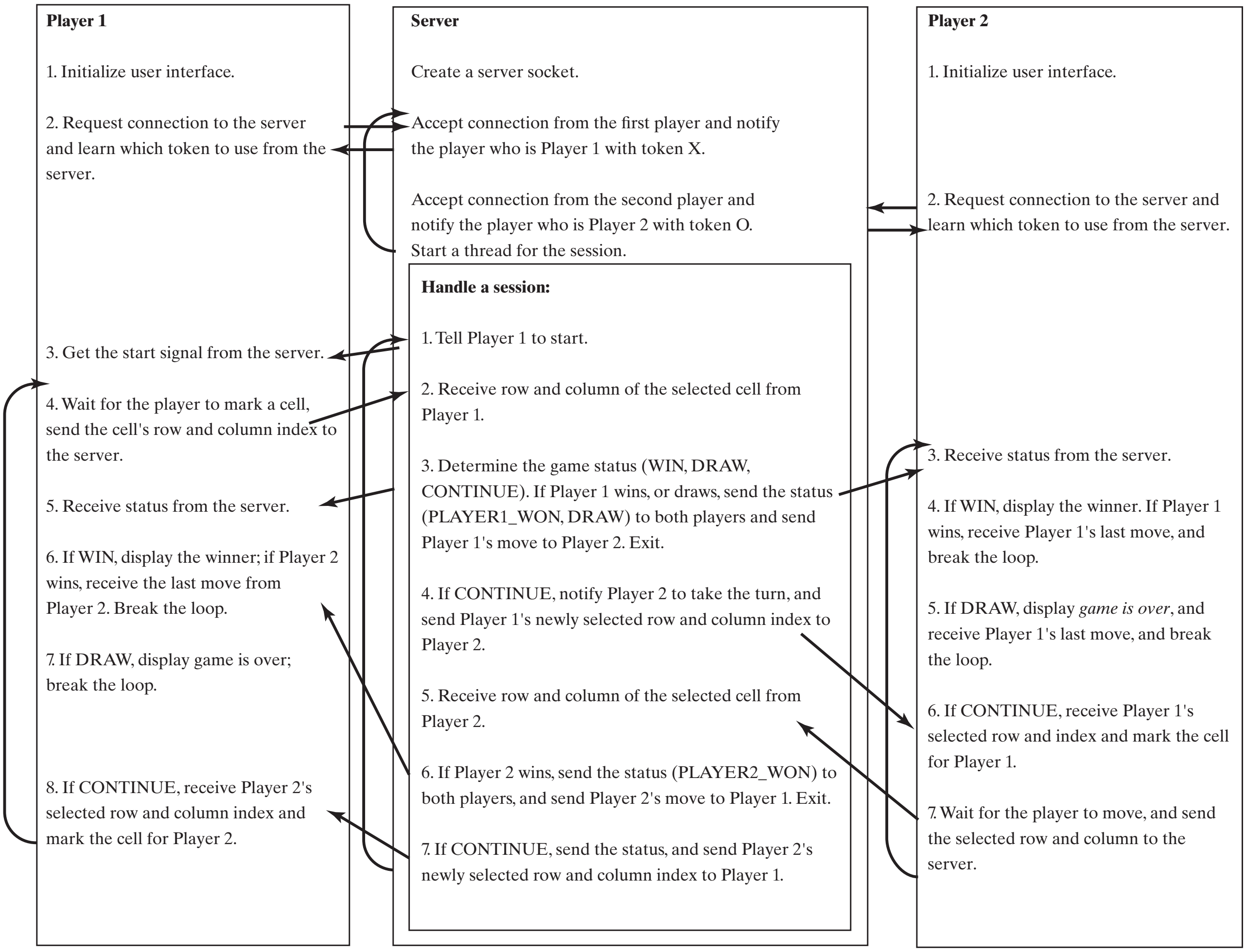
FIGURE 33.12 The server can create many sessions, each of which facilitates a tic-tac-toe game for two players.

The server does not have to be a graphical component, but creating it in a GUI in which game information can be viewed is user friendly. You can create a scroll pane to hold a text area in the GUI and display game information in the text area. The server creates a thread to handle a game session when two players are connected to the server.

The client is responsible for interacting with the players. It creates a user interface with nine cells and displays the game title and status to the players in the labels. The client class is very similar to the `TicTacToe` class presented in the case study in Listing 16.13. However, the client in this example does not determine the game status (win or draw); it simply passes the moves to the server and receives the game status from the server.

Based on the foregoing analysis, you can create the following classes:

- `TicTacToeServer` serves all the clients in Listing 33.9.
- `HandleASession` facilitates the game for two players. This class is defined in Listing 33.9, TicTacToeServer.java.



Player 1

1. Initialize user interface.

2. Request connection to the server and learn which token to use from the server.

3. Get the start signal from the server.

4. Wait for the player to mark a cell, send the cell's row and column index to the server.

5. Receive status from the server.

6. If WIN, display the winner; if Player 2 wins, receive the last move from Player 2. Break the loop.

7. If DRAW, display game is over; break the loop.

8. If CONTINUE, receive Player 2's selected row and column index and mark the cell for Player 2.

Server

Create a server socket.

Accept connection from the first player and notify the player who is Player 1 with token X.

Accept connection from the second player and notify the player who is Player 2 with token O. Start a thread for the session.

Handle a session:

1. Tell Player 1 to start.

2. Receive row and column of the selected cell from Player 1.

3. Determine the game status (WIN, DRAW, CONTINUE). If Player 1 wins, or draws, send the status (PLAYER1_WON, DRAW) to both players and send Player 1's move to Player 2. Exit.

4. If CONTINUE, notify Player 2 to take the turn, and send Player 1's newly selected row and column index to Player 2.

5. Receive row and column of the selected cell from Player 2.

6. If Player 2 wins, send the status (PLAYER2_WON) to both players, and send Player 2's move to Player 1. Exit.

7. If CONTINUE, send the status, and send Player 2's newly selected row and column index to Player 1.

Player 2

1. Initialize user interface.

2. Request connection to the server and learn which token to use from the server.

3. Receive status from the server.

4. If WIN, display the winner. If Player 1 wins, receive Player 1's last move, and break the loop.

5. If DRAW, display *game is over*, and receive Player 1's last move, and break the loop.

6. If CONTINUE, receive Player 1's selected row and index and mark the cell for Player 1.

7. Wait for the player to move, and send the selected row and column to the server.

Figure 33.13 The server starts a thread to facilitate communications between the two players.

- `TicTacToeClient` models a player in Listing 33.10.
- `Cell` models a cell in the game. It is an inner class in `TicTacToeClient`.
- `TicTacToeConstants` is an interface that defines the constants shared by all the classes in the example in Listing 33.8.

The relationships of these classes are shown in Figure 33.14.

Listing 33.8 TicTacToeConstants.java

```java
public interface TicTacToeConstants {
  public static int PLAYER1 = 1; // Indicate player 1
  public static int PLAYER2 = 2; // Indicate player 2
  public static int PLAYER1_WON = 1; // Indicate player 1 won
  public static int PLAYER2_WON = 2; // Indicate player 2 won
  public static int DRAW = 3; // Indicate a draw
  public static int CONTINUE = 4; // Indicate to continue
}
```

Listing 33.9 TicTacToeServer.java

```java
import java.io.*;
import java.net.*;
```

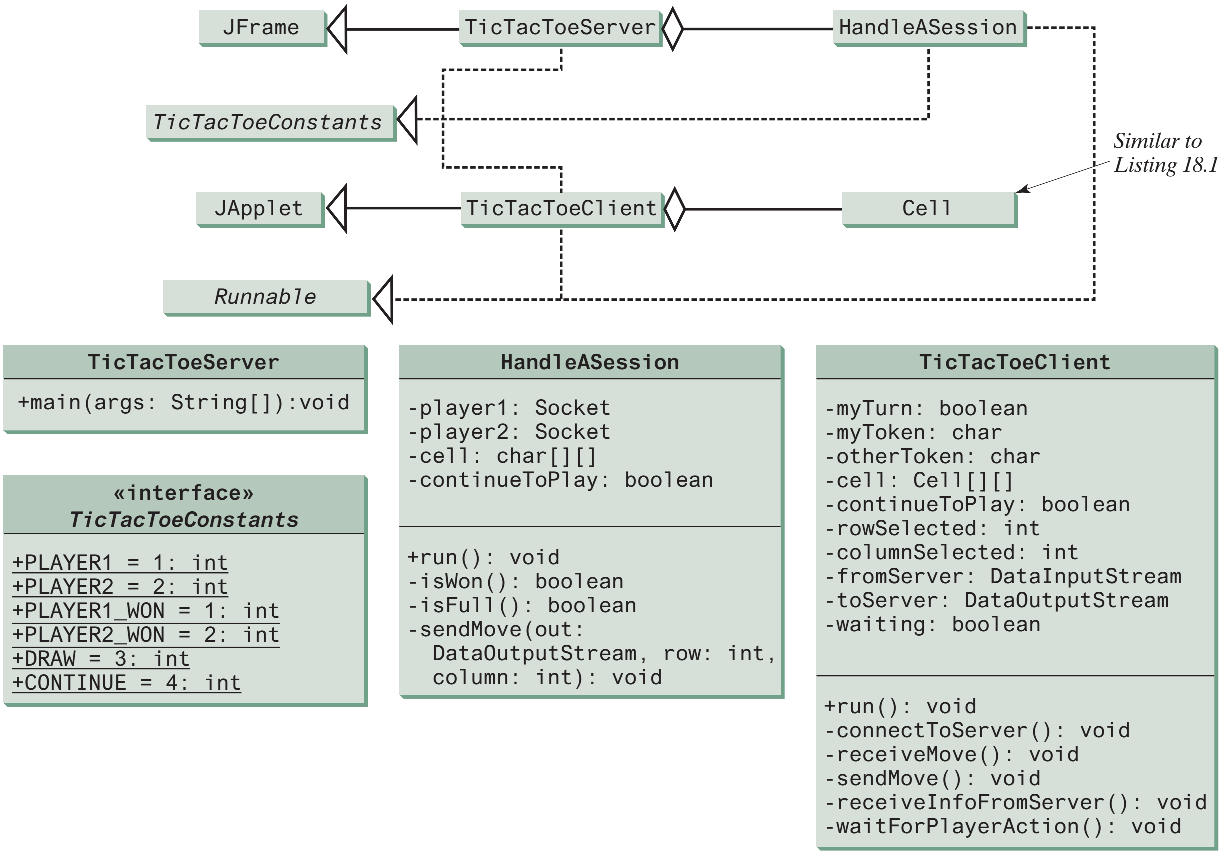


FIGURE 33.14 TicTacToeServer creates an instance of HandleASession for each session of two players. TicTacToeClient creates nine cells in the UI.

```
import java.util.Date;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.stage.Stage;

public class TicTacToeServer extends Application
    implements TicTacToeConstants {
  private int sessionNo = 1; // Number a session

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    TextArea taLog = new TextArea();

    // Create a scene and place it in the stage
    Scene scene = new Scene(new ScrollPane(taLog), 450, 200);
    primaryStage.setTitle("TicTacToeServer"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    new Thread( () -> {
      try {
        // Create a server socket
```

create UI

```
     ServerSocket serverSocket = new ServerSocket(8000);
     Platform.runLater(() -> taLog.appendText(new Date() +
      ": Server started at socket 8000\n"));

     // Ready to create a session for every two players
     while (true) {
      Platform.runLater(() -> taLog.appendText(new Date() +
       ": Wait for players to join session " + sessionNo + '\n'));

      // Connect to player 1
      Socket player1 = serverSocket.accept();

      Platform.runLater(() -> {
       taLog.appendText(new Date() + ": Player 1 joined session "
        + sessionNo + '\n');
       taLog.appendText("Player 1's IP address" +
        player1.getInetAddress().getHostAddress() + '\n');
      });

      // Notify that the player is Player 1
      new DataOutputStream(
       player1.getOutputStream()).writeInt(PLAYER1);

      // Connect to player 2
      Socket player2 = serverSocket.accept();

      Platform.runLater(() -> {
       taLog.appendText(new Date() +
        ": Player 2 joined session " + sessionNo + '\n');
       taLog.appendText("Player 2's IP address" +
        player2.getInetAddress().getHostAddress() + '\n');
      });

      // Notify that the player is Player 2
      new DataOutputStream(
       player2.getOutputStream()).writeInt(PLAYER2);

      // Display this session and increment session number
      Platform.runLater(() ->
       taLog.appendText(new Date() +
        ": Start a thread for session " + sessionNo++ + '\n'));

      // Launch a new thread for this session of two players
      new Thread(new HandleASession(player1, player2)).start();
     }
    }
    catch(IOException ex) {
     ex.printStackTrace();
    }
   }).start();
  }

  // Define the thread class for handling a new session for two players
  class HandleASession implements Runnable, TicTacToeConstants {
   private Socket player1;
   private Socket player2;

   // Create and initialize cells
   private char[][] cell = new char[3][3];

   private DataInputStream fromPlayer1;
```

server socket

connect to client

to player1

connect to client

to player2

a session for two players

```
    private DataOutputStream toPlayer1;
    private DataInputStream fromPlayer2;
    private DataOutputStream toPlayer2;

    // Continue to play
    private boolean continueToPlay = true;

    /** Construct a thread */
    public HandleASession(Socket player1, Socket player2) {
      this.player1 = player1;
      this.player2 = player2;

      // Initialize cells
      for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
          cell[i][j] = ' ';
    }

    /** Implement the run() method for the thread */
    public void run() {
      try {
        // Create data input and output streams
```

IO streams

```
        DataInputStream fromPlayer1 = new DataInputStream(
          player1.getInputStream());
        DataOutputStream toPlayer1 = new DataOutputStream(
          player1.getOutputStream());
        DataInputStream fromPlayer2 = new DataInputStream(
          player2.getInputStream());
        DataOutputStream toPlayer2 = new DataOutputStream(
          player2.getOutputStream());

        // Write anything to notify player 1 to start
        // This is just to let player 1 know to start
        toPlayer1.writeInt(1);

        // Continuously serve the players and determine and report
        // the game status to the players
        while (true) {
          // Receive a move from player 1
          int row = fromPlayer1.readInt();
          int column = fromPlayer1.readInt();
          cell[row][column] = 'X';

          // Check if Player 1 wins
```

X won?

```
          if (isWon('X')) {
            toPlayer1.writeInt(PLAYER1_WON);
            toPlayer2.writeInt(PLAYER1_WON);
            sendMove(toPlayer2, row, column);
            break; // Break the loop
          }
```

Is full?

```
          else if (isFull()) { // Check if all cells are filled
            toPlayer1.writeInt(DRAW);
            toPlayer2.writeInt(DRAW);
            sendMove(toPlayer2, row, column);
            break;
          }
          else {
            // Notify player 2 to take the turn
            toPlayer2.writeInt(CONTINUE);

          // Send player 1's selected row and column to player 2
          sendMove(toPlayer2, row, column);
        }

        // Receive a move from Player 2
        row = fromPlayer2.readInt();
        column = fromPlayer2.readInt();
        cell[row][column] = 'O';

        // Check if Player 2 wins
        if (isWon('O')) {
          toPlayer1.writeInt(PLAYER2_WON);
          toPlayer2.writeInt(PLAYER2_WON);
          sendMove(toPlayer1, row, column);
          break;
        }
        else {
          // Notify player 1 to take the turn
          toPlayer1.writeInt(CONTINUE);

          // Send player 2's selected row and column to player 1
          sendMove(toPlayer1, row, column);
        }
      }
    }
    catch(IOException ex) {
      ex.printStackTrace();
    }
  }

  /** Send the move to other player */
  private void sendMove(DataOutputStream out, int row, int column)
      throws IOException {
    out.writeInt(row); // Send row index
    out.writeInt(column); // Send column index
  }

  /** Determine if the cells are all occupied */
  private boolean isFull() {
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
        if (cell[i][j] == ' ')
          return false; // At least one cell is not filled

    // All cells are filled
    return true;
  }

  /** Determine if the player with the specified token wins */
  private boolean isWon(char token) {
    // Check all rows
    for (int i = 0; i < 3; i++)
      if ((cell[i][0] == token)
          && (cell[i][1] == token)
          && (cell[i][2] == token)) {
        return true;
      }

    /** Check all columns */
    for (int j = 0; j < 3; j++)
```

O won?

send a move

```
        if ((cell[0][j] == token)
            && (cell[1][j] == token)
            && (cell[2][j] == token)) {
          return true;
      }

      /** Check major diagonal */
      if ((cell[0][0] == token)
          && (cell[1][1] == token)
          && (cell[2][2] == token)) {
        return true;
      }

      /** Check subdiagonal */
      if ((cell[0][2] == token)
          && (cell[1][1] == token)
          && (cell[2][0] == token)) {
        return true;
      }

      /** All checked, but no winner */
      return false;
    }
  }
}
```

Listing 33.10 TicTacToeClient.java

```
import java.io.*;
import java.net.*;
import java.util.Date;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Ellipse;
import javafx.scene.shape.Line;
import javafx.stage.Stage;

public class TicTacToeClient extends Application
    implements TicTacToeConstants {
  // Indicate whether the player has the turn
  private boolean myTurn = false;

  // Indicate the token for the player
  private char myToken = ' ';

  // Indicate the token for the other player
  private char otherToken = ' ';

  // Create and initialize cells
  private Cell[][] cell = new Cell[3][3];

  // Create and initialize a title label
  private Label lblTitle = new Label();

  // Create and initialize a status label
  private Label lblStatus = new Label();

  // Indicate selected row and column by the current move
  private int rowSelected;
  private int columnSelected;

  // Input and output streams from/to server
  private DataInputStream fromServer;
  private DataOutputStream toServer;

  // Continue to play?
  private boolean continueToPlay = true;

  // Wait for the player to mark a cell
  private boolean waiting = true;

  // Host name or ip
  private String host = "localhost";

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Pane to hold cell
    GridPane pane = new GridPane();
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
        pane.add(cell[i][j] = new Cell(i, j), j, i);

    BorderPane borderPane = new BorderPane();
    borderPane.setTop(lblTitle);
    borderPane.setCenter(pane);
    borderPane.setBottom(lblStatus);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 320, 350);
    primaryStage.setTitle("TicTacToeClient"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    // Connect to the server
    connectToServer();
  }

  private void connectToServer() {
    try {
      // Create a socket to connect to the server
      Socket socket = new Socket(host, 8000);

      // Create an input stream to receive data from the server
      fromServer = new DataInputStream(socket.getInputStream());

      // Create an output stream to send data to the server
      toServer = new DataOutputStream(socket.getOutputStream());
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
```

create UI

connect to server

input from server

output to server

```

    // Control the game on a separate thread
    new Thread(() -> {
      try {
        // Get notification from the server
        int player = fromServer.readInt();

        // Am I player 1 or 2?
        if (player == PLAYER1) {
          myToken = 'X';
          otherToken = 'O';
          Platform.runLater(() -> {
            lblTitle.setText("Player 1 with token 'X'");
            lblStatus.setText("Waiting for player 2 to join");
          });

          // Receive startup notification from the server
          fromServer.readInt(); // Whatever read is ignored

          // The other player has joined
          Platform.runLater(() ->
            lblStatus.setText("Player 2 has joined. I start first"));

          // It is my turn
          myTurn = true;
        }
        else if (player == PLAYER2) {
          myToken = 'O';
          otherToken = 'X';
          Platform.runLater(() -> {
            lblTitle.setText("Player 2 with token 'O'");
            lblStatus.setText("Waiting for player 1 to move");
          });
        }

        // Continue to play
        while (continueToPlay) {
          if (player == PLAYER1) {
            waitForPlayerAction(); // Wait for player 1 to move
            sendMove(); // Send the move to the server
            receiveInfoFromServer(); // Receive info from the server
          }
          else if (player == PLAYER2) {
            receiveInfoFromServer(); // Receive info from the server
            waitForPlayerAction(); // Wait for player 2 to move
            sendMove(); // Send player 2's move to the server
          }
        }
      }
      catch (Exception ex) {
        ex.printStackTrace();
      }
    }).start();
  }

  /** Wait for the player to mark a cell */
  private void waitForPlayerAction() throws InterruptedException {
    while (waiting) {
      Thread.sleep(100);
    }

    waiting = true;
  }

  /** Send this player's move to the server */
  private void sendMove() throws IOException {
    toServer.writeInt(rowSelected); // Send the selected row
    toServer.writeInt(columnSelected); // Send the selected column
  }

  /** Receive info from the server */
  private void receiveInfoFromServer() throws IOException {
    // Receive game status
    int status = fromServer.readInt();

    if (status == PLAYER1_WON) {
      // Player 1 won, stop playing
      continueToPlay = false;
      if (myToken == 'X') {
        Platform.runLater(() -> lblStatus.setText("I won! (X)"));
      }
      else if (myToken == 'O') {
        Platform.runLater(() ->
          lblStatus.setText("Player 1 (X) has won!"));
        receiveMove();
      }
    }
    else if (status == PLAYER2_WON) {
      // Player 2 won, stop playing
      continueToPlay = false;
      if (myToken == 'O') {
        Platform.runLater(() -> lblStatus.setText("I won! (O)"));
      }
      else if (myToken == 'X') {
        Platform.runLater(() ->
          lblStatus.setText("Player 2 (O) has won!"));
        receiveMove();
      }
    }
    else if (status == DRAW) {
      // No winner, game is over
      continueToPlay = false;
      Platform.runLater(() ->
        lblStatus.setText("Game is over, no winner!"));

      if (myToken == 'O') {
        receiveMove();
      }
    }
    else {
      receiveMove();
      Platform.runLater(() -> lblStatus.setText("My turn"));
      myTurn = true; // It is my turn
    }
  }

  private void receiveMove() throws IOException {
    // Get the other player's move
    int row = fromServer.readInt();
    int column = fromServer.readInt();
```

```
      Platform.runLater(() -> cell[row][column].setToken(otherToken));
    }

    // An inner class for a cell
```

model a cell

```
    public class Cell extends Pane {
      // Indicate the row and column of this cell in the board
      private int row;
      private int column;

      // Token used for this cell
      private char token = ' ';

      public Cell(int row, int column) {
        this.row = row;
        this.column = column;
        this.setPrefSize(2000, 2000); // What happens without this?
        setStyle("-fx-border-color: black"); // Set cell's border
```

register listener

```
        this.setOnMouseClicked(e -> handleMouseClick());
      }

      /** Return token */
      public char getToken() {
        return token;
      }

      /** Set a new token */
      public void setToken(char c) {
        token = c;
        repaint();
      }

      protected void repaint() {
        if (token == 'X') {
```

draw X

```
          Line line1 = new Line(10, 10,
            this.getWidth() - 10, this.getHeight() - 10);
          line1.endXProperty().bind(this.widthProperty().subtract(10));
          line1.endYProperty().bind(this.heightProperty().subtract(10));
          Line line2 = new Line(10, this.getHeight() - 10,
            this.getWidth() - 10, 10);
          line2.startYProperty().bind(
            this.heightProperty().subtract(10));
          line2.endXProperty().bind(this.widthProperty().subtract(10));

          // Add the lines to the pane
          this.getChildren().addAll(line1, line2);
        }
        else if (token == 'O') {
```

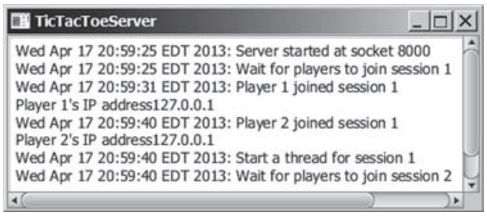
draw O

```
          Ellipse ellipse = new Ellipse(this.getWidth() / 2,
            this.getHeight() / 2, this.getWidth() /  2 - 10,
            this.getHeight() / 2 - 10);
          ellipse.centerXProperty().bind(
            this.widthProperty().divide(2));
          ellipse.centerYProperty().bind(
            this.heightProperty().divide(2));
          ellipse.radiusXProperty().bind(
            this.widthProperty().divide(2).subtract(10));
          ellipse.radiusYProperty().bind(
            this.heightProperty().divide(2).subtract(10));
          ellipse.setStroke(Color.BLACK);
          ellipse.setFill(Color.WHITE);
```

```

        getChildren().add(ellipse); // Add the ellipse to the pane
      }
    }

    /* Handle a mouse click event */
    private void handleMouseClick() {
      // If cell is not occupied and the player has the turn
      if (token == ' ' && myTurn) {
        setToken(myToken); // Set the player's token in the cell
        myTurn = false;
        rowSelected = row;
        columnSelected = column;
        lblStatus.setText("Waiting for the other player to move");
        waiting = false; // Just completed a successful move
      }
    }
  }
}
```

mouse clicked handler

The server can serve any number of sessions simultaneously. Each session takes care of two players. The client can be deployed to run as a Java applet. To run a client as a Java applet from a Web browser, the server must run from a Web server. Figures 33.15 and 33.16 show sample runs of the server and the clients.



```
Wed Apr 17 20:59:25 EDT 2013: Server started at socket 8000
Wed Apr 17 20:59:25 EDT 2013: Wait for players to join session 1
Wed Apr 17 20:59:31 EDT 2013: Player 1 joined session 1
Player 1's IP address127.0.0.1
Wed Apr 17 20:59:40 EDT 2013: Player 2 joined session 1
Player 2's IP address127.0.0.1
Wed Apr 17 20:59:40 EDT 2013: Start a thread for session 1
Wed Apr 17 20:59:40 EDT 2013: Wait for players to join session 2
```

FIGURE 33.15 TicTacToeServer accepts connection requests and creates sessions to serve pairs of players.

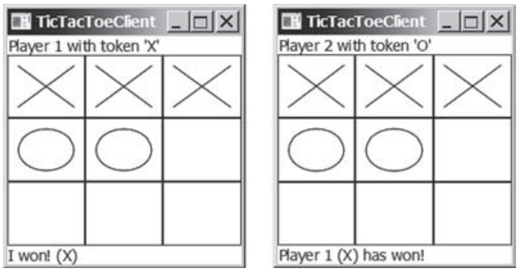


FIGURE 33.16 TicTacToeClient can run as an applet or standalone.

The TicTacToeConstants interface defines the constants shared by all the classes in the project. Each class that uses the constants needs to implement the interface. Centrally defining constants in an interface is a common practice in Java.

Once a session is established, the server receives moves from the players in alternation. Upon receiving a move from a player, the server determines the status of the game. If the game is not finished, the server sends the status (CONTINUE) and the player's move to the other player. If the game is won or a draw, the server sends the status (`PLAYER1_WON`, `PLAYER2_WON`, or `DRAW`) to both players.

The implementation of Java network programs at the socket level is tightly synchronized. An operation to send data from one machine requires an operation to receive data from the other machine. As shown in this example, the server and the client are tightly synchronized to send or receive data.

Check Point

33.6.1 What would happen if the preferred size for a cell is not set in line 227 in Listing 33.10?

33.6.2 If a player does not have the turn but clicks on an empty cell, what will the client program in Listing 33.10 do?

Key Terms

client socket	33-3	packet-based communication	33-2
domain name	33-2	server socket	33-2
domain name server	33-2	socket	33-2
localhost	33-3	stream-based communication	33-2
IP address	33-2	TCP	33-2
port	33-2	UDP	33-2

Chapter Summary

1. Java supports stream sockets and datagram sockets. *Stream sockets* use TCP (Transmission Control Protocol) for data transmission, whereas *datagram sockets* use UDP (User Datagram Protocol). Since TCP can detect lost transmissions and resubmit them, transmissions are lossless and reliable. UDP, in contrast, cannot guarantee lossless transmission.

2. To create a server, you must first obtain a server socket, using `new ServerSocket(port)`. After a server socket is created, the server can start to listen for connections, using the `accept()` method on the server socket. The client requests a connection to a server by using `new Socket(serverName, port)` to create a client socket.

3. Stream socket communication is very much like input/output stream communication after the connection between a server and a client is established. You can obtain an input stream using the `getInputStream()` method and an output stream using the `getOutputStream()` method on the socket.

4. A server must often work with multiple clients at the same time. You can use threads to handle the server's multiple clients simultaneously by creating a thread for each connection.

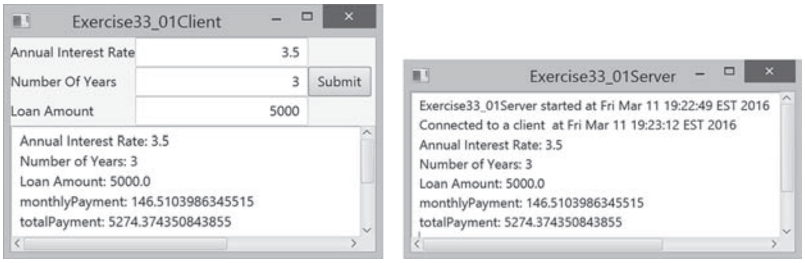
Quiz

Answer the quiz for this chapter online at book Companion Website.

Programming Exercises

Section 33.2

***33.1** (*Loan server*) Write a server for a client. The client sends loan information (annual interest rate, number of years, and loan amount) to the server (see Figure 33.17a). The server computes monthly payment and total payment, and sends them back to the client (see Figure 33.17b). Name the client Exercise33_01Client and the server Exercise33_01Server.



Annual Interest Rate 3.5
Number Of Years 3
Loan Amount 5000

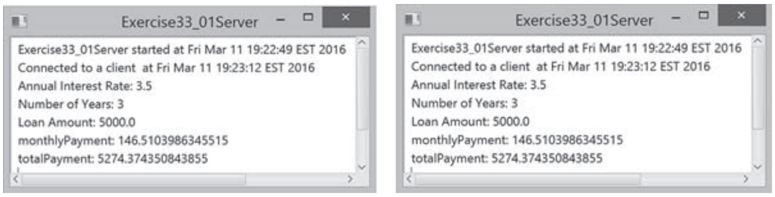
```
Annual Interest Rate: 3.5
Number of Years: 3
Loan Amount: 5000.0
monthlyPayment: 146.5103986345515
totalPayment: 5274.374350843855
```

```
Exercise33_01Server started at Fri Mar 11 19:22:49 EST 2016
Connected to a client  at Fri Mar 11 19:23:12 EST 2016
Annual Interest Rate: 3.5
Number of Years: 3
Loan Amount: 5000.0
monthlyPayment: 146.5103986345515
totalPayment: 5274.374350843855
```

(a) (b)

Figure 33.17 The client in (a) sends the annual interest rate, number of years, and loan amount to the server and receives the monthly payment and total payment from the server in (b).

***33.2** (*BMI server*) Write a server for a client. The client sends the weight and height for a person to the server (see Figure 33.18a). The server computes BMI (Body Mass Index) and sends back to the client a string that reports the BMI (see Figure 33.18b). See Section 3.8 for computing BMI. Name the client Exercise33_02Client and the server Exercise33_02Server.



```
Exercise33_01Server started at Fri Mar 11 19:22:49 EST 2016
Connected to a client  at Fri Mar 11 19:23:12 EST 2016
Annual Interest Rate: 3.5
Number of Years: 3
Loan Amount: 5000.0
monthlyPayment: 146.5103986345515
totalPayment: 5274.374350843855
```

```
Exercise33_01Server started at Fri Mar 11 19:22:49 EST 2016
Connected to a client  at Fri Mar 11 19:23:12 EST 2016
Annual Interest Rate: 3.5
Number of Years: 3
Loan Amount: 5000.0
monthlyPayment: 146.5103986345515
totalPayment: 5274.374350843855
```

(a) (b)

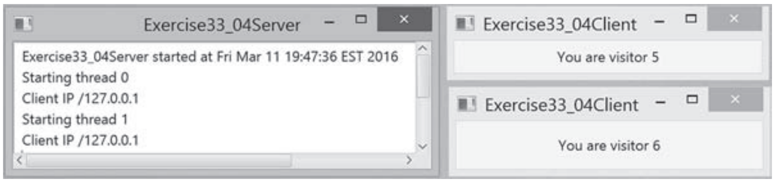
Figure 33.18 The client in (a) sends the weight and height of a person to the server and receives the BMI from the server in (b).

Sections 33.3 and 33.4

***33.3** (*Loan server for multiple clients*) Revise Programming Exercise 33.1 to write a server for multiple clients.

Section 33.5

33.4 (*Count clients*) Write a server that tracks the number of the clients connected to the server. When a new connection is established, the count is incremented by 1. The count is stored using a random-access file. Write a client program that receives the count from the server and displays a message, such as "You are visitor number 11", as shown in Figure 33.19. Name the client Exercise33_04Client and the server Exercise33_04Server.



```
Exercise33_04Server started at Fri Mar 11 19:47:36 EST 2016
Starting thread 0
Client IP /127.0.0.1
Starting thread 1
Client IP /127.0.0.1
```

You are visitor 5

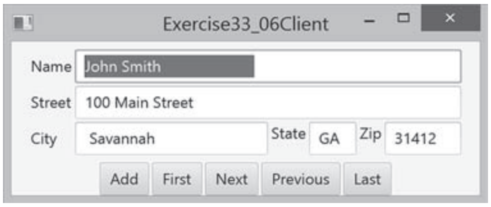
You are visitor 6

FIGURE 33.19 The client displays how many times the server has been accessed. The server stores the count.

33.5 (*Send loan information in an object*) Revise Exercise 33.1 for the client to send a loan object that contains annual interest rate, number of years, and loan amount and for the server to send the monthly payment and total payment.

Section 33.6

33.6 (*Display and add addresses*) Develop a client/server application to view and add addresses, as shown in Figure 33.20.



Name: John Smith
Street: 100 Main Street
City: Savannah State: GA Zip: 31412

FIGURE 33.20 You can view and add an address.

- Use the `StudentAddress` class defined in Listing 33.5 to hold the name, street, city, state, and zip in an object.
- The user can use the buttons *First*, *Next*, *Previous*, and *Last* to view an address, and the *Add* button to add a new address.
- Limit the concurrent connections to two clients.

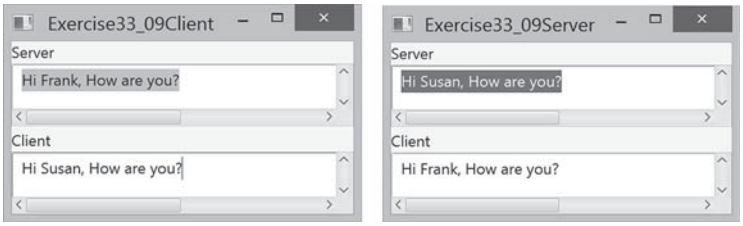
Name the client Exercise33_06Client and the server Exercise33_6Server.

***33.7** (*Transfer last 100 numbers in an array*) Programming Exercise 22.12 retrieves the last 100 prime numbers from a file **PrimeNumbers.dat**. Write a client program that requests the server to send the last 100 prime numbers in an array. Name the server program Exercise33_07Server and the client program Exercise33_07Client. Assume the numbers of the `long` type are stored in **PrimeNumbers.dat** in binary format.

***33.8** (*Transfer last 100 numbers in an `ArrayList`*) Programming Exercise 24.12 retrieves the last 100 prime numbers from a file **PrimeNumbers.dat**. Write a client program that requests the server to send the last 100 prime numbers in an `ArrayList`. Name the server program Exercise33_08Server and the client program Exercise33_08Client. Assume the numbers of the `long` type are stored in **PrimeNumbers.dat** in binary format.

Section 33.7

****33.9** (*Chat*) Write a program that enables two users to chat. Implement one user as the server (see Figure 33.21a) and the other as the client (see Figure 33.21b). The server has two text areas: one for entering text, and the other (noneditable) for displaying text received from the client. When the user presses the *Enter* key, the current line is sent to the client. The client has two text areas: one (noneditable) for displaying text from the server and the other for entering text. When the user presses the *Enter* key, the current line is sent to the server. Name the client Exercise33_09Client and the server Exercise33_09Server.



Hi Frank, How are you?

Hi Susan, How are you?

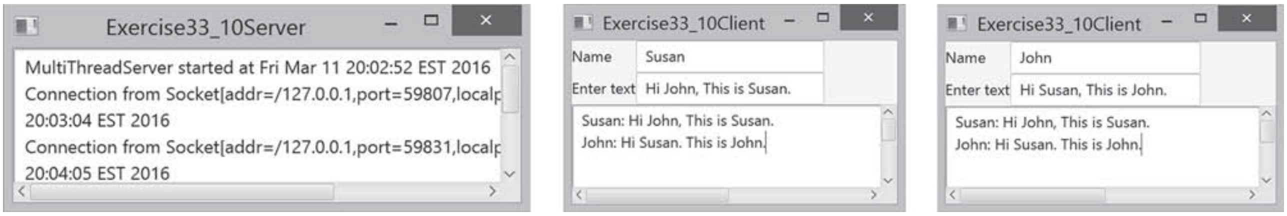
Hi Susan, How are you?

Hi Frank, How are you?

(a) (b)

FIGURE 33.21 The server and client send text to and receive text from each other.

*****33.10** (*Multiple client chat*) Write a program that enables any number of clients to chat. Implement one server that serves all the clients, as shown in Figure 33.22. Name the client Exercise33_10Client and the server Exercise33_10Server.



```
MultiThreadServer started at Fri Mar 11 20:02:52 EST 2016
Connection from Socket[addr=/127.0.0.1,port=59807,localp
20:03:04 EST 2016
Connection from Socket[addr=/127.0.0.1,port=59831,localp
20:04:05 EST 2016
```

Name: Susan

Enter text: Hi John, This is Susan.

```
Susan: Hi John, This is Susan.
John: Hi Susan. This is John.
```

Name: John

Enter text: Hi Susan, This is John.

```
Susan: Hi John, This is Susan.
John: Hi Susan. This is John.
```

(a) (b) (c)

FIGURE 33.22 The server starts in (a) with three clients in (b) and (c).

CHAPTER

34

Java Database Programming

Objectives

- To understand the concepts of databases and database management systems (§34.2).
- To understand the relational data model: relational data structures, constraints, and languages (§34.2).
- To use SQL to create and drop tables and to retrieve and modify data (§34.3).
- To learn how to load a driver, connect to a database, execute statements, and process result sets using JDBC (§34.4).
- To use prepared statements to execute precompiled SQL statements (§34.5).
- To use callable statements to execute stored SQL procedures and functions (§34.6).
- To explore database metadata using the `DatabaseMetaData` and `ResultSetMetaData` interfaces (§34.7).

34.1 Introduction

Key Point

Java provides the API for developing database applications that works with any relational database systems.

You may have heard a lot about database systems. Database systems are everywhere. Your social security information is stored in a database by the government. If you shop online, your purchase information is stored in a database by the company. If you attend a university, your academic information is stored in a database by the university. Database systems not only store data, they also provide means of accessing, updating, manipulating, and analyzing data. Your social security information is updated periodically, and you can register for courses online. Database systems play an important role in society and in commerce.

This chapter introduces database systems, the SQL language, and how database applications can be developed using Java. If you already know SQL, you can skip Sections 34.2 and 34.3.

34.2 Relational Database Systems

Key Point

SQL is the standard database language for defining and accessing databases.

database system

A *database system* consists of a database, the software that stores and manages data in the database, and the application programs that present data and enable the user to interact with the database system, as shown in Figure 34.1.

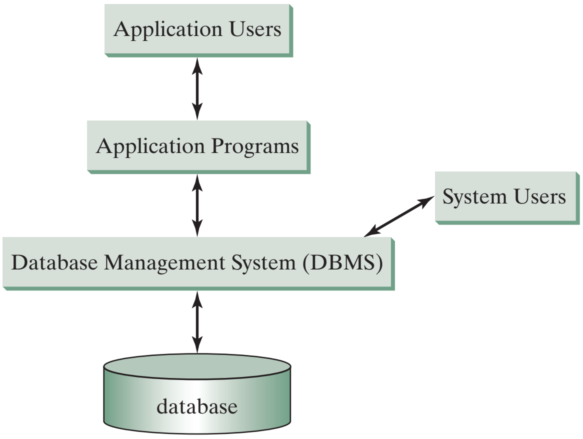


FIGURE 34.1 A database system consists of data, database management software, and application programs.

A *database* is a repository of data that form information. When you purchase a database system—such as MySQL, Oracle, IBM's DB2 and Informix, Microsoft SQL Server, or Sybase—from a software vendor, you actually purchase the software comprising a *database management system* (DBMS). Database management systems are designed for use by professional programmers and are not suitable for ordinary customers. Application programs are built on top of the DBMS for customers to access and update the database. Thus, application programs can be viewed as the *interfaces* between the database system and its users. Application programs may be stand-alone GUI applications or Web applications and may access several different database systems in the network, as shown in Figure 34.2.

DBMS

Most of today's database systems are *relational database systems.* They are based on the relational data model, which has three key components: structure, integrity, and language.

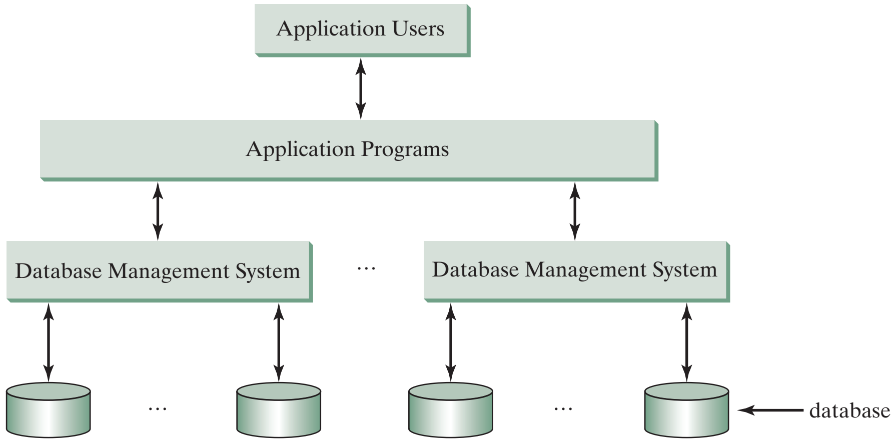


Figure 34.2 An application program can access multiple database systems.

Structure defines the representation of the data. *Integrity* imposes constraints on the data. *Language* provides the means for accessing and manipulating data.

34.2.1 Relational Structures

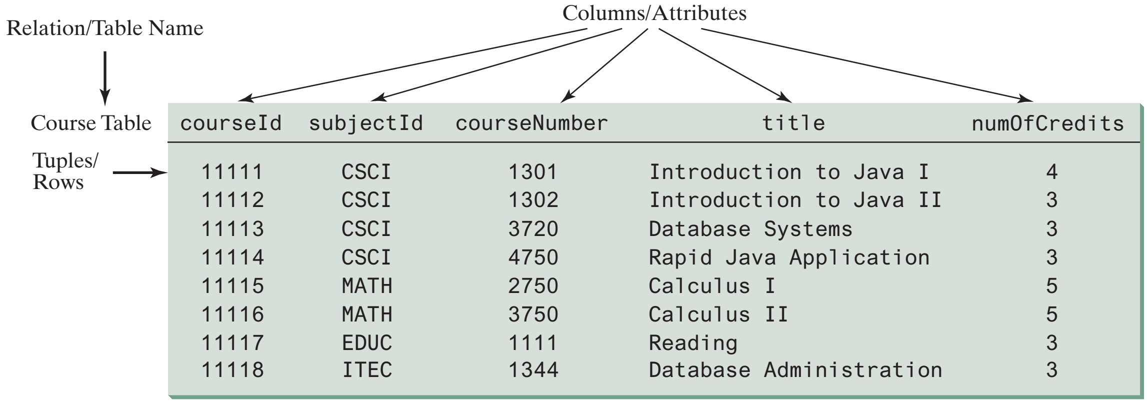
relational model

The relational model is built around a simple and natural structure. A *relation* is actually a table that consists of nonduplicate rows. Tables are easy to understand and use. The relational model provides a simple yet powerful way to represent data.

tuple

attribute

A row of a table represents a *record*, and a column of a table represents the *value of a single attribute* of the record. In relational database theory, a row is called a *tuple*, and a column is called an *attribute*. Figure 34.3 shows a sample table that stores information about the courses offered by a university. The table has eight tuples, and each tuple has five attributes.


Relation/Table Name
Columns/Attributes
Course Table
Tuples/
Rows

courseId	subjectId	courseNumber	title	numOfCredits
11111	CSCI	1301	Introduction to Java I	4
11112	CSCI	1302	Introduction to Java II	3
11113	CSCI	3720	Database Systems	3
11114	CSCI	4750	Rapid Java Application	3
11115	MATH	2750	Calculus I	5
11116	MATH	3750	Calculus II	5
11117	EDUC	1111	Reading	3
11118	ITEC	1344	Database Administration	3

Figure 34.3 A table has a table name, column names, and rows.

Tables describe the relationship among data. Each row in a table represents a record of related data. For example, "11111," "CSCI," "1301," "Introduction to Java I," and "4" are related to form a record (the first row in Figure 34.3) in the `Course` table. Just as the data in the same row are related, so too data in different tables may be related through common attributes. Suppose the database has two other tables, `Student` and `Enrollment`, as shown in

Figures 34.4 and 34.5. The Course table and the Enrollment table are related through their common attribute courseId, and the Enrollment table and the Student table are related through ssn.

Student Table

ssn	firstName	mi	lastName	phone	birthDate	street	zipCode	deptID
444111110	Jacob	R	Smith	9129219434	1985-04-09	99 Kingston Street	31435	BIOL
444111111	John	K	Stevenson	9129219434	null	100 Main Street	31411	BIOL
444111112	George	K	Smith	9129213454	1974-10-10	1200 Abercorn St.	31419	CS
444111113	Frank	E	Jones	9125919434	1970-09-09	100 Main Street	31411	BIOL
444111114	Jean	K	Smith	9129219434	1970-02-09	100 Main Street	31411	CHEM
444111115	Josh	R	Woo	7075989434	1970-02-09	555 Franklin St.	31411	CHEM
444111116	Josh	R	Smith	9129219434	1973-02-09	100 Main Street	31411	BIOL
444111117	Joy	P	Kennedy	9129229434	1974-03-19	103 Bay Street	31412	CS
444111118	Toni	R	Peterson	9129229434	1964-04-29	103 Bay Street	31412	MATH
444111119	Patrick	R	Stoneman	9129229434	1969-04-29	101 Washington St.	31435	MATH
444111120	Rick	R	Carter	9125919434	1986-04-09	19 West Ford St.	31411	BIOL

Figure 34.4 A Student table stores student information.

Enrollment Table

ssn	courseId	dateRegistered	grade
444111110	11111	2004-03-19	A
444111110	11112	2004-03-19	B
444111110	11113	2004-03-19	C
444111111	11111	2004-03-19	D
444111111	11112	2004-03-19	F
444111111	11113	2004-03-19	A
444111112	11114	2004-03-19	B
444111112	11115	2004-03-19	C
444111112	11116	2004-03-19	D
444111113	11111	2004-03-19	A
444111113	11113	2004-03-19	A
444111114	11115	2004-03-19	B
444111115	11115	2004-03-19	F
444111115	11116	2004-03-19	F
444111116	11111	2004-03-19	D
444111117	11111	2004-03-19	D
444111118	11111	2004-03-19	A
444111118	11112	2004-03-19	D
444111118	11113	2004-03-19	B

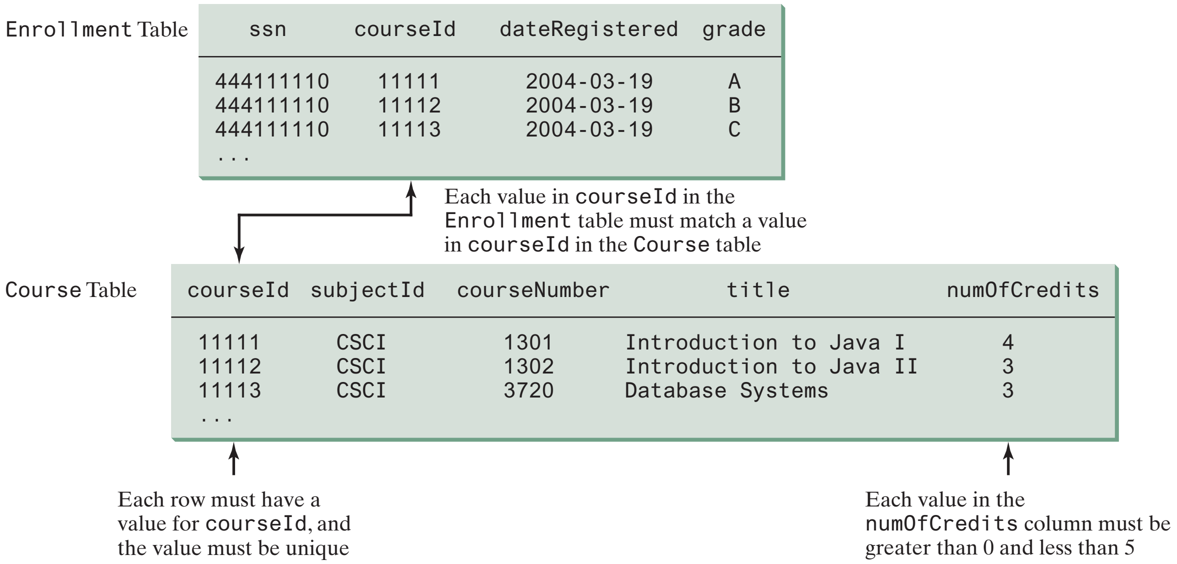
Figure 34.5 An Enrollment table stores student enrollment information.

34.2.2 Integrity Constraints

integrity constraint

An *integrity constraint* imposes a condition that all the legal values in a table must satisfy. Figure 34.6 shows an example of some integrity constraints in the Subject and Course tables.

In general, there are three types of constraints: domain constraints, primary key constraints, and foreign key constraints. *Domain constraints* and *primary key constraints* are known as *intrarelational constraints*, meaning that a constraint involves only one relation. The *foreign key constraint* is *interrelational*, meaning that a constraint involves more than one relation.



Enrollment Table

ssn	courseId	dateRegistered	grade
444111110	11111	2004-03-19	A
444111110	11112	2004-03-19	B
444111110	11113	2004-03-19	C
...			

Each value in `courseId` in the `Enrollment` table must match a value in `courseId` in the `Course` table

Course Table

courseId	subjectId	courseNumber	title	numOfCredits
11111	CSCI	1301	Introduction to Java I	4
11112	CSCI	1302	Introduction to Java II	3
11113	CSCI	3720	Database Systems	3
...				

Each row must have a value for `courseId`, and the value must be unique

Each value in the `numOfCredits` column must be greater than 0 and less than 5

Figure 34.6 The `Enrollment` table and the `Course` table have integrity constraints.

Domain Constraints

domain constraint

Domain constraints specify the permissible values for an attribute. Domains can be specified using standard data types, such as integers, floating-point numbers, fixed-length strings, and variant-length strings. The standard data type specifies a broad range of values. Additional constraints can be specified to narrow the ranges. For example, you can specify that the `numOfCredits` attribute (in the `Course` table) must be greater than 0 and less than 5. If an attribute has different values for each tuple in a relation, you can specify the attribute to be unique. You can also specify whether an attribute can be `null`, which is a special value in a database meaning unknown or not applicable. As shown in the `Student` table, `birthDate` may be `null`.

Primary Key Constraints

superkey

A primary key is a set of attributes that uniquely identifyies the tuples in a relations. Why is it called a primary key, rather than simply key? To understand this, it is helpful to know superkeys, keys, and candidate keys. A *superkey* is an attribute or a set of attributes that uniquely identifies the relation. That is, no two tuples have the same values on a superkey. By definition, a relation consists of a set of distinct tuples. The set of all attributes in the relation forms a superkey.

candidate key

primary key

A *key* `K` is a minimal superkey, meaning that any proper subset of `K` is not a superkey. A relation can have several keys. In this case, each of the keys is called a *candidate key*. The *primary key* is one of the candidate keys designated by the database designer. The primary key is often used to identify tuples in a relation. As shown in Figure 34.6, `courseId` is the primary key in the `Course` table, and `ssn` and `courseId` form a primary key in the `Enrollment` table.

Foreign Key Constraints

relational database

In a *relational database*, data are related. Tuples in a relation are related, and tuples in different relations are related through their common attributes. Informally speaking, the common attributes are foreign keys. The *foreign key constraints* define the relationships among relations.

foreign key constraint

foreign key

Formally, a set of attributes *FK* is a *foreign key* in a relation *R* that references relation *T* if it satisfies the following two rules:

- The attributes in *FK* have the same domain as the primary key in *T*.
- A nonnull value on *FK* in *R* must match a primary key value in *T*.

As shown in Figure 34.6, `courseId` is the foreign key in `Enrollment` that references the primary key `courseId` in `Course`. Every `courseId` value must match a `courseId` value in `Course`.

Enforcing Integrity Constraints

auto enforcement

The database management system enforces integrity constraints and rejects operations that would violate them. For example, if you attempt to insert the new record ("11115," "CSCI," "2490," "C++ Programming," "0") into the `Course` table, it would fail because the credit hours must be greater than `0`; if you attempted to insert a record with the same primary key as an existing record in the table, the DBMS would report an error and reject the operation; if you attempted to delete a record from the `Course` table whose primary key value is referenced by the records in the `Enrollment` table, the DBMS would reject this operation.

Note

All relational database systems support primary key constraints and foreign key constraints, but not all database systems support domain constraints. In the Microsoft Access database, for example, you cannot specify the constraint that `numOfCredits` is greater than `0` and less than `5`.

Check Point

34.2.1 What are superkeys, candidate keys, and primary keys?

34.2.2 What is a foreign key?

34.2.3 Can a relation have more than one primary key or foreign key?

34.2.4 Does a foreign key need to be a primary key in the same relation?

34.2.5 Does a foreign key need to have the same name as its referenced primary key?

34.2.6 Can a foreign key value be null?

34.3 SQL

Key Point

Structured Query Language (SQL) is the language for defining tables and integrity constraints, and for accessing and manipulating data.

SQL

database language

SQL (pronounced "S-Q-L" or "sequel") is the universal language for accessing relational database systems. Application programs may allow users to access a database without directly using SQL, but these applications themselves must use SQL to access the database. This section introduces some basic SQL commands.

Note

standard SQL

There are many relational database management systems. They share the common SQL language but do not all support every feature of SQL. Some systems have their own extensions to SQL. This section introduces standard SQL supported by all systems.

SQL can be used on MySQL, Oracle, Sybase, IBM DB2, IBM Informix, MS Access, Apache Derby, or any other relational database system. Apache Derby is an open source relational database management system developed using Java. Oracle distributes Apache Derby as Java DB and bundled with Java so you can use it in any Java application without installing a database. Java DB is ideal for supporting a small database in a Java application. This chapter uses MySQL to demonstrate SQL and Java database programming.

The Companion Website contains the following supplements on how to install and use three popular databases: MySQL, Oracle, and Java DB:

MySQL Tutorial
- Supplement IV.B: Tutorial for MySQL

Oracle Tutorial
- Supplement IV.C: Tutorial for Oracle

Java DB Tutorial
- Supplement IV.D: Tutorial for Java DB

34.3.1 Creating a User Account on MySQL

Assume you have installed MySQL 5 with the default configuration. To match all the examples in this book, you should create a user named *scott* with the password *tiger*. You can perform the administrative tasks using the MySQL Workbench or using the command line. MySQL Workbench is a GUI tool for managing MySQL databases. Here are the steps to create a user from the command line:

1. From the DOS command prompt, type

   ```
   mysql –uroot -p
   ```

   You will be prompted to enter the root password, as shown in Figure 34.7.

2. At the mysql prompt, enter

   ```
   use mysql;
   ```

3. To create user **scott** with password **tiger**, enter

   ```
   create user 'scott'@'localhost' identified by 'tiger';
   ```

4. To grant privileges to **scott**, enter

   ```
   grant select, insert, update, delete, create, create view, drop,
      execute, references on *.* to 'scott'@'localhost';
   ```

   - If you want to enable remote access of the account from any IP address, enter

     ```
     grant all privileges on *.* to 'scott'@'%'
       identified by 'tiger';
     ```

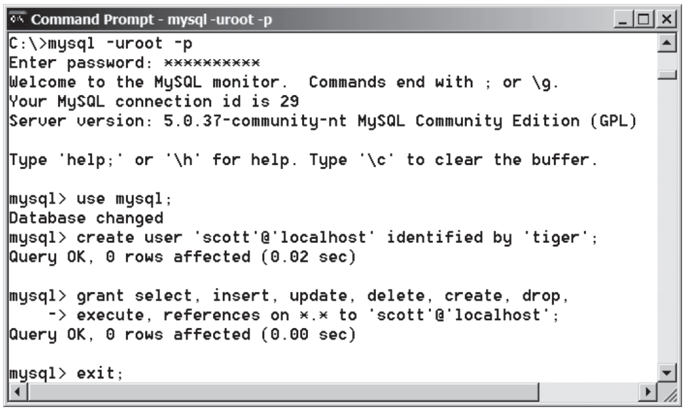
   - If you want to restrict the account's remote access to just one particular IP address, enter

     ```
     grant all privileges on *.* to 'scott'@'ipAddress'
       identified by 'tiger';
     ```

5. Enter

   ```
   exit;
   ```

   to exit the MySQL console.



```
C:\>mysql -uroot -p
Enter password: **********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 29
Server version: 5.0.37-community-nt MySQL Community Edition (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use mysql;
Database changed
mysql> create user 'scott'@'localhost' identified by 'tiger';
Query OK, 0 rows affected (0.02 sec)

mysql> grant select, insert, update, delete, create, drop,
    -> execute, references on *.* to 'scott'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> exit;
```

FIGURE 34.7 You can access a MySQL database server from the command window.

stop mysql
start mysql

Note
On Windows, your MySQL database server starts every time your computer starts. You can stop it by typing the command `net stop mysql` and restart it by typing the command `net start mysql`.

By default, the server contains two databases named **mysql** and **test**. The **mysql** database contains the tables that store information about the server and its users. This database is intended for the server administrator to use. For example, the administrator can use it to create users and grant or revoke user privileges. Since you are the owner of the server installed on your system, you have full access to the **mysql** database. However, you should not create user tables in the mysql database. You can use the **test** database to store data or create new databases. You can also create a new database using the command `create database` ***`databasename`*** or delete an existing database using the command `drop database` ***`databasename`***.

34.3.2 Creating a Database

To match the examples in this book, you should create a database named **javabook**. Here are the steps to create it:

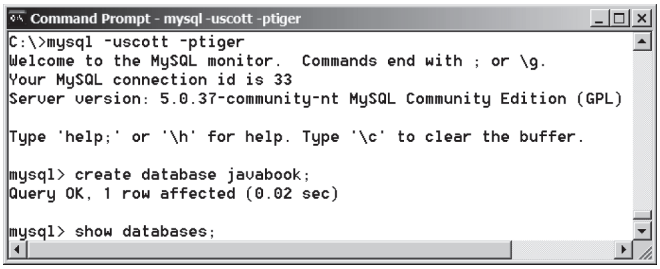
1. From the DOS command prompt, type

   ```
   mysql –uscott -ptiger
   ```

   to login to mysql, as shown in Figure 34.8.

2. At the mysql prompt, enter

   ```
   create database javabook;
   ```



```
C:\>mysql -uscott -ptiger
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 33
Server version: 5.0.37-community-nt MySQL Community Edition (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> create database javabook;
Query OK, 1 row affected (0.02 sec)

mysql> show databases;
```

Figure 34.8 You can create databases in MySQL.

For your convenience, the SQL statements for creating and initializing tables used in this book are provided in Supplement IV.A. You can download the script for MySQL and save it to **script.sql**. To execute the script, first switch to the **javabook** database using the following command:

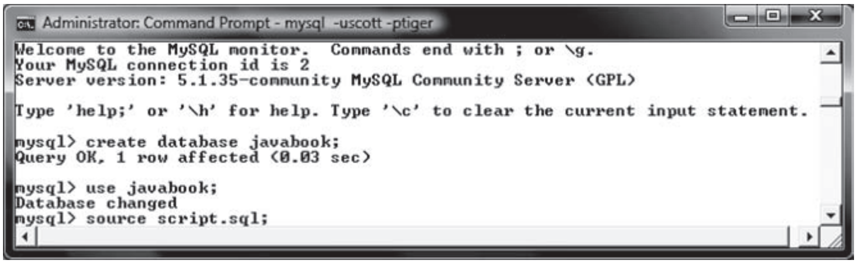
```
use javabook;
```

then type

run script file

```
source script.sql;
```

as shown in Figure 34.9.



```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.1.35-community MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> create database javabook;
Query OK, 1 row affected (0.03 sec)

mysql> use javabook;
Database changed
mysql> source script.sql;
```

Figure 34.9 You can run SQL commands in a script file.

Note

You can populate the **javabook** database using the script from Supplement IV.A.

populating database

34.3.3 Creating and Dropping Tables

create table

Tables are the essential objects in a database. To create a table, use the `create table` statement to specify a table name, attributes, and types, as in the following example:

```
create table Course (
  courseId char(5),
  subjectId char(4) not null,
  courseNumber integer,
  title varchar(50) not null,
  numOfCredits integer,
  primary key (courseId)
);
```

This statement creates the `Course` table with attributes `courseId`, `subjectId`, `courseNumber`, `title`, and `numOfCredits`. Each attribute has a data type that specifies the type of data stored in the attribute. `char(5)` specifies that `courseId` consists of five characters. `varchar(50)` specifies that `title` is a variant-length string with a maximum of 50 characters. `integer` specifies that `courseNumber` is an integer. The primary key is `courseId`.

The tables `Student` and `Enrollment` can be created as follows:

```
create table Student (
  ssn char(9),
  firstName varchar(25),
  mi char(1),
  lastName varchar(25),
  birthDate date,
  street varchar(25),
  phone char(11),
  zipCode char(5),
  deptId char(4),
  primary key (ssn)
);
```

```
create table Enrollment (
  ssn char(9),
  courseId char(5),
  dateRegistered date,
  grade char(1),
  primary key (ssn, courseId),
  foreign key (ssn) references
    Student(ssn),
  foreign key (courseId) references
    Course(courseId)
);
```

Note

naming convention

SQL keywords are not case sensitive. This book adopts the following naming conventions: tables are named in the same way as Java classes, and attributes are named in the same way as Java variables. SQL keywords are named in the same way as Java keywords.

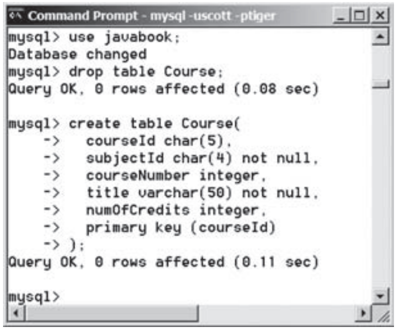
drop table

If a table is no longer needed, it can be dropped permanently using the `drop table` command. For example, the following statement drops the `Course` table:

```
drop table Course;
```

If a table to be dropped is referenced by other tables, you have to drop the other tables first. For example, if you have created the tables `Course`, `Student`, and `Enrollment` and want to drop `Course`, you have to first drop `Enrollment`, because `Course` is referenced by `Enrollment`.

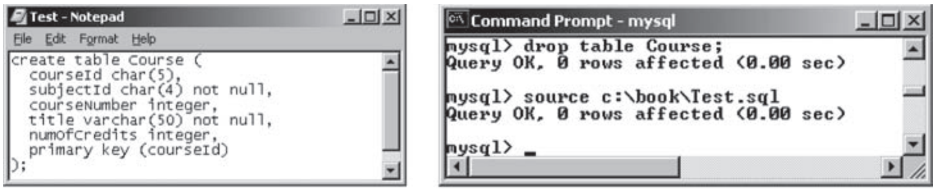
Figure 34.10 shows how to enter the `create table` statement from the MySQL console.



```
mysql> use javabook;
Database changed
mysql> drop table Course;
Query OK, 0 rows affected (0.08 sec)

mysql> create table Course(
    ->   courseId char(5),
    ->   subjectId char(4) not null,
    ->   courseNumber integer,
    ->   title varchar(50) not null,
    ->   numOfCredits integer,
    ->   primary key (courseId)
    -> );
Query OK, 0 rows affected (0.11 sec)

mysql>
```

FIGURE 34.10 A table is created using the `create table` statement.

If you make typing errors, you have to retype the whole command. To avoid retyping, you can save the command in a file, then run the command from the file. To do so, create a text file to contain commands, named, for example, **test.sql**. You can create the text file using any text editor, such as Notepad, as shown in Figure 34.11a. To comment a line, precede it with two dashes. You can now run the script file by typing `source test.sql` from the SQL command prompt, as shown in Figure 34.11b.



```
create table Course (
  courseId char(5),
  subjectId char(4) not null,
  courseNumber integer,
  title varchar(50) not null,
  numOfCredits integer,
  primary key (courseId)
);
```

(a)

```
mysql> drop table Course;
Query OK, 0 rows affected (0.00 sec)

mysql> source c:\book\Test.sql
Query OK, 0 rows affected (0.00 sec)

mysql> _
```

(b)

FIGURE 34.11 (a) You can use Notepad to create a text file for SQL commands. (b) You can run the SQL commands in a script file from MySQL.

34.3.4 Simple Insert, Update, and Delete

Once a table is created, you can insert data into it. You can also update and delete records. This section introduces simple insert, update, and delete statements.

The syntax to insert a record into a table is:

```
insert into tableName [(column1, column2, ..., column)]
values (value1, value2, ..., valuen);
```

For example, the following statement inserts a record into the Course table. The new record has the courseId '11113', subjectId 'CSCI', courseNumber '3720', title 'Database Systems', and creditHours 3.

```
insert into Course (courseId, subjectId, courseNumber, title, numOfCredits)
values ('11113', 'CSCI', '3720', 'Database Systems', 3);
```

The column names are optional. If they are omitted, all the column values for the record must be entered, even though the columns have default values. String values are case sensitive and enclosed inside single quotation marks in SQL.

The syntax to update a table is:

```
update tableName
set column1 = newValue1 [, column2 = newValue2, ...]
[where condition];
```

For example, the following statement changes the numOfCredits for the course whose title is Database Systems to 4.

```
update Course
set numOfCredits = 4
where title = 'Database Systems';
```

The syntax to delete records from a table is:

```
delete from tableName
[where condition];
```

For example, the following statement deletes the Database Systems course from the Course table:

```
delete from Course
where title = 'Database Systems';
```

The following statement deletes all the records from the Course table:

```
delete from Course;
```

34.3.5 Simple Queries

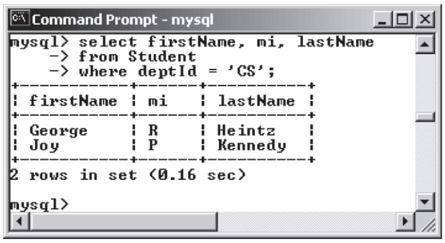
To retrieve information from tables, use a select statement with the following syntax:

```
select column-list
from table-list
[where condition];
```

The select clause lists the columns to be selected. The from clause refers to the tables involved in the query. The optional where clause specifies the conditions for the selected rows.

Query 1: Select all the students in the CS department, as shown in Figure 34.12.

```
select firstName, mi, lastName
from Student
where deptId = 'CS';
```



```
mysql> select firstName, mi, lastName
    -> from Student
    -> where deptId = 'CS';
+-----------+------+----------+
| firstName | mi   | lastName |
+-----------+------+----------+
| George    | R    | Heintz   |
| Joy       | P    | Kennedy  |
+-----------+------+----------+
2 rows in set (0.16 sec)

mysql>
```

FIGURE 34.12 The result of the `select` statement is displayed in the MySQL console.

34.3.6 Comparison and Boolean Operators

SQL has six comparison operators, as shown in Table 34.1, and three Boolean operators, as shown in Table 34.2.

TABLE 34.1 Comparison Operators

Operator	*Description*
`=`	Equal to
`<>` or `!=`	Not equal to
`<`	Less than
`<=`	Less than or equal to
`>`	Greater than
`>=`	Greater than or equal to

TABLE 34.2 Boolean Operators

Operator	*Description*
`not`	Logical negation
`and`	Logical conjunction
`or`	Logical disjunction

Note

The comparison and Boolean operators in SQL have the same meanings as in Java. In SQL the `equal to` operator is `=`, but in Java it is `==`. In SQL the `not equal to` operator is `<>` or `!=`, but in Java it is `!=`. The `not`, `and`, and `or` operators are `!`, `&&` (`&`), and `||` (`|`) in Java.

Query 2: Get the names of the students who are in the CS dept and live in the ZIP code 31411.

```
select firstName, mi, lastName
from Student
where deptId = 'CS' and zipCode = '31411';
```

Note

To select all the attributes from a table, you don't have to list all the attribute names in the select clause. Instead, you can just use an *asterisk* (*), which stands for all the attributes. For example, the following query displays all the attributes of the students who are in the CS dept and live in ZIP code 31411.

```
select *
from Student
where deptId = 'CS' and zipCode = '31411';
```

34.3.7 The like, between-and, and is null Operators

SQL has a like operator that can be used for pattern matching. The syntax to check whether a string s has a pattern p is

s like p or s not like p

You can use the wildcard characters % (percent symbol) and _ (underline symbol) in the pattern p. % matches zero or more characters, and _ matches any single character in s. For example, lastName like '_mi%' matches any string whose second and third letters are m and i. lastName not like '_mi%' excludes any string whose second and third letters are m and i.

Note

In earlier versions of MS Access, the wildcard character is *, and the character ? matches any single character.

The between-and operator checks whether a value v is between two other values, v1 and v2, using the following syntax:

v between v1 and v2 or v not between v1 and v2

v between v1 and v2 is equivalent to v >= v1 and v <= v2, and v not between v1 and v2 is equivalent to v < v1 or v > v2.

The is null operator checks whether a value v is null using the following syntax:

v is null or v is not null

Query 3: Get the Social Security numbers of the students whose grades are between ‘C’ and ‘A’.

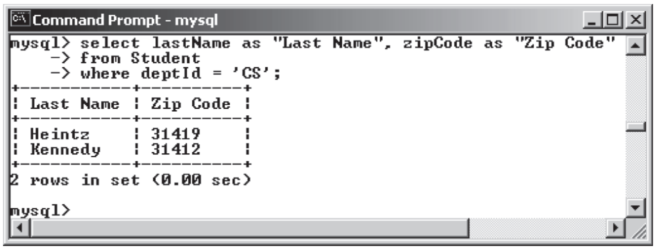
```
select ssn
from Enrollment
where grade between 'C' and 'A';
```

34.3.8 Column Alias

When a query result is displayed, SQL uses the column names as column headings. Usually the user gives abbreviated names for the columns, and the columns cannot have spaces when the table is created. Sometimes it is desirable to give more descriptive names in the result heading. You can use the column aliases with the following syntax:

```
select columnName [as] alias
```

Query 4: Get the last name and ZIP code of the students in the CS department. Display the column headings as “Last Name” for lastName and “Zip Code” for zipCode. The query result is shown in Figure 34.13.



```
mysql> select lastName as "Last Name", zipCode as "Zip Code"
    -> from Student
    -> where deptId = 'CS';
+-----------+----------+
| Last Name | Zip Code |
+-----------+----------+
| Heintz    | 31419    |
| Kennedy   | 31412    |
+-----------+----------+
2 rows in set (0.00 sec)

mysql>
```

FIGURE 34.13 You can use a column alias in the display.

```
select lastName as "Last Name", zipCode as "Zip Code"
from Student
where deptId = 'CS';
```

Note

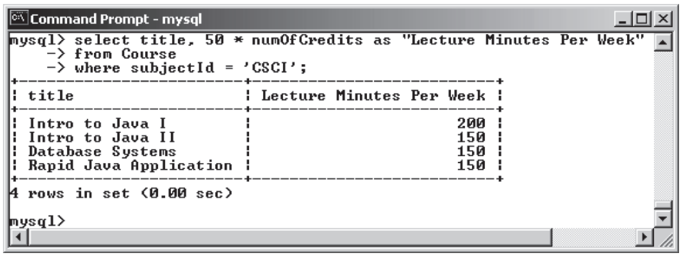
The **as** keyword is optional in MySQL and Oracle, but it is required in MS Access.

34.3.9 The Arithmetic Operators

You can use the arithmetic operators * (multiplication), / (division), + (addition), and − (subtraction) in SQL.

Query 5: Assume a credit hour is 50 minutes of lectures and get the total minutes for each course with the subject CSCI. The query result is shown in Figure 34.14.

```
select title, 50 * numOfCredits as "Lecture Minutes Per Week"
from Course
where subjectId = 'CSCI';
```



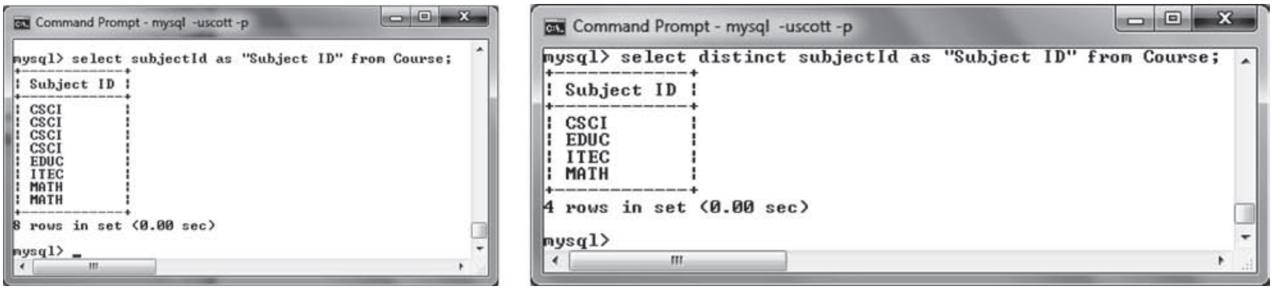
```
mysql> select title, 50 * numOfCredits as "Lecture Minutes Per Week"
    -> from Course
    -> where subjectId = 'CSCI';
+-------------------------+--------------------------+
| title                   | Lecture Minutes Per Week |
+-------------------------+--------------------------+
| Intro to Java I         |                      200 |
| Intro to Java II        |                      150 |
| Database Systems        |                      150 |
| Rapid Java Application  |                      150 |
+-------------------------+--------------------------+
4 rows in set (0.00 sec)

mysql>
```

FIGURE 34.14 You can use arithmetic operators in SQL.

34.3.10 Displaying Distinct Tuples

SQL provides the `distinct` keyword, which can be used to eliminate duplicate tuples in the result. Figure 34.15a displays all the subject IDs used by the courses, and Figure 34.15b displays all the distinct subject IDs used by the courses using the following statement:

```
select distinct subjectId as "Subject ID"
from Course;
```



Command Prompt - mysql -uscott -p

```
mysql> select subjectId as "Subject ID" from Course;
+------------+
| Subject ID |
+------------+
| CSCI       |
| CSCI       |
| CSCI       |
| CSCI       |
| EDUC       |
| ITEC       |
| MATH       |
| MATH       |
+------------+
8 rows in set (0.00 sec)

mysql> _
```

(a)

Command Prompt - mysql -uscott -p

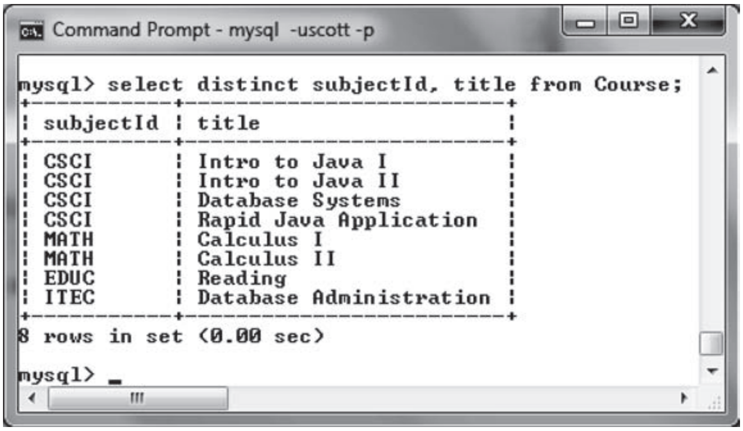
```
mysql> select distinct subjectId as "Subject ID" from Course;
+------------+
| Subject ID |
+------------+
| CSCI       |
| EDUC       |
| ITEC       |
| MATH       |
+------------+
4 rows in set (0.00 sec)

mysql>
```

(b)

FIGURE 34.15 (a) The duplicate tuples are displayed. (b) The distinct tuples are displayed.

When there is more than one column in the `select` clause, the `distinct` keyword applies to the whole tuple in the result. For example, the following statement displays all tuples with distinct `subjectId` and `title`, as shown in Figure 34.16. Note some tuples may have the same `subjectId` but different `title`. These tuples are distinct.

```
select distinct subjectId, title
from Course;
```



```
mysql> select distinct subjectId, title from Course;
+-----------+----------------------------+
| subjectId | title                      |
+-----------+----------------------------+
| CSCI      | Intro to Java I            |
| CSCI      | Intro to Java II           |
| CSCI      | Database Systems           |
| CSCI      | Rapid Java Application     |
| MATH      | Calculus I                 |
| MATH      | Calculus II                |
| EDUC      | Reading                    |
| ITEC      | Database Administration    |
+-----------+----------------------------+
8 rows in set (0.00 sec)

mysql> _
```

FIGURE 34.16 The keyword `distinct` applies to the entire tuple.

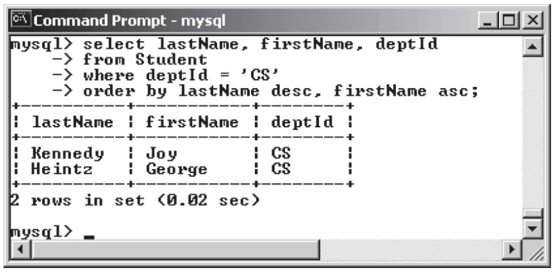
34.3.11 Displaying Sorted Tuples

SQL provides the `order by` clause to sort the output using the following syntax:

```
select column-list
from table-list
[where condition]
[order by columns-to-be-sorted];
```

In the syntax, `columns-to-be-sorted` specifies a column or a list of columns to be sorted. By default, the order is ascending. To sort in a descending order, append the `desc` keyword. You could also append the `asc` keyword after `columns-to-be-sorted`, but it is not necessary. When multiple columns are specified, the rows are sorted based on the first column, then the rows with the same values on the first column are sorted based on the second column, and so on.

Query 6: List the full names of the students in the CS department, ordered primarily on their last names in descending order and secondarily on their first names in ascending order. The query result is shown in Figure 34.17.



```
mysql> select lastName, firstName, deptId
    -> from Student
    -> where deptId = 'CS'
    -> order by lastName desc, firstName asc;
+----------+-----------+--------+
| lastName | firstName | deptId |
+----------+-----------+--------+
| Kennedy  | Joy       | CS     |
| Heintz   | George    | CS     |
+----------+-----------+--------+
2 rows in set (0.02 sec)

mysql> _
```

FIGURE 34.17 You can sort results using the `order by` clause.

```
select lastName, firstName, deptId
from Student
where deptId = 'CS'
order by lastName desc, firstName asc;
```

34.3.12 Joining Tables

Often you need to get information from multiple tables, as demonstrated in the next query.

Query 7: List the courses taken by the student Jacob Smith. To solve this query, you need to join tables `Student` and `Enrollment`, as shown in Figure 34.18.

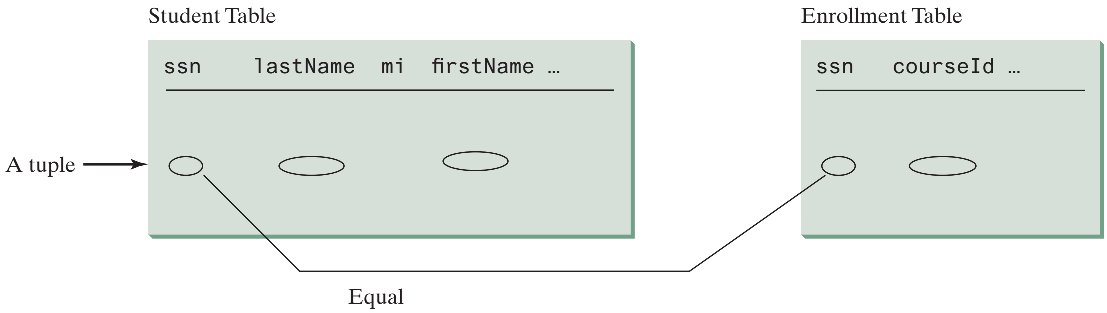
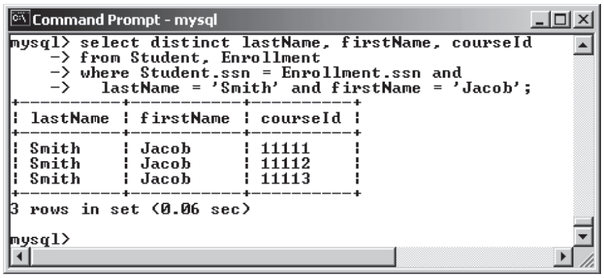


FIGURE 34.18 `Student` and `Enrollment` are joined on `ssn`.

You can write the query in SQL as follows:

```
select distinct lastName, firstName, courseId
from Student, Enrollment
where Student.ssn = Enrollment.ssn and
  lastName = 'Smith' and firstName = 'Jacob';
```

The tables `Student` and `Enrollment` are listed in the `from` clause. The query examines every pair of rows, each made of one item from `Student` and another from `Enrollment` and selects the pairs that satisfy the condition in the `where` clause. The rows in `Student` have the last name, Smith, and the first name, Jacob, and both rows from `Student` and `Enrollment` have the same `ssn` values. For each pair selected, `lastName` and `firstName` from `Student` and `courseId` from `Enrollment` are used to produce the result, as shown in Figure 34.19. `Student` and `Enrollment` have the same attribute `ssn`. To distinguish them in a query, use `Student.ssn` and `Enrollment.ssn`.



```
mysql> select distinct lastName, firstName, courseId
    -> from Student, Enrollment
    -> where Student.ssn = Enrollment.ssn and
    ->   lastName = 'Smith' and firstName = 'Jacob';
+----------+-----------+----------+
| lastName | firstName | courseId |
+----------+-----------+----------+
| Smith    | Jacob     | 11111    |
| Smith    | Jacob     | 11112    |
| Smith    | Jacob     | 11113    |
+----------+-----------+----------+
3 rows in set (0.06 sec)

mysql>
```

FIGURE 34.19 Query 7 demonstrates queries involving multiple tables.

For more features of SQL, see Supplements IV.H and IV.I.

Check Point

34.3.1 Create the tables `Course`, `Student`, and `Enrollment` using the `create table` statements in Section 34.3.3, Creating and Dropping Tables. Insert rows into the `Course`, `Student`, and `Enrollment` tables using the data in Figures 34.3–34.5.

34.3.2 List all CSCI courses with at least four credit hours.

34.3.3 List all students whose last names contain the letter *e* two times.

34.3.4 List all students whose birthdays are null.

34.3.5 List all students who take Math courses.

34.3.6 List the number of courses in each subject.

34.3.7 Assume each credit hour is 50 minutes of lectures. Get the total minutes for the courses that each student takes.

34.4 JDBC

Key Point

JDBC is the Java API for accessing relational database.

The Java API for developing Java database applications is called *JDBC*. JDBC is the trademarked name of a Java API that supports Java programs that access relational databases. JDBC is not an acronym, but it is often thought to stand for Java Database Connectivity.

JDBC provides Java programmers with a uniform interface for accessing and manipulating relational databases. Using the JDBC API, applications written in the Java programming language can execute SQL statements, retrieve results, present data in a user-friendly interface, and propagate changes back to the database. The JDBC API can also be used to interact with multiple data sources in a distributed, heterogeneous environment.

The relationships among Java programs, JDBC API, JDBC drivers, and relational databases are shown in Figure 34.20. The JDBC API is a set of Java interfaces and classes used to write Java programs for accessing and manipulating relational databases. Since a JDBC driver serves as the interface to facilitate communications between JDBC and a proprietary database, JDBC drivers are database specific and are normally provided by the database vendors. You need

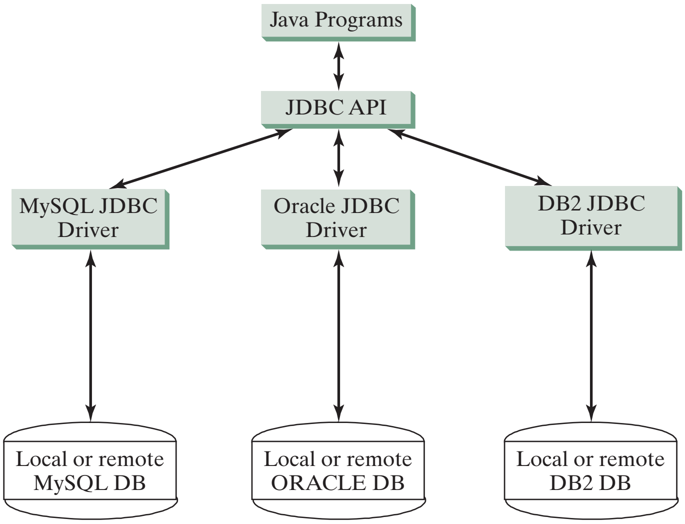


FIGURE 34.20 Java programs access and manipulate databases through JDBC drivers.

MySQL JDBC drivers to access the MySQL database, Oracle JDBC drivers to access the Oracle database, and DB2 JDBC driver to access the DB2 database.

34.4.1 Developing Database Applications Using JDBC

The JDBC API is a Java application program interface to generic SQL databases that enables Java developers to develop DBMS-independent Java applications using a uniform interface.

The JDBC API consists of classes and interfaces for establishing connections with databases, sending SQL statements to databases, processing the results of SQL statements, and obtaining database metadata. Four key interfaces are needed to develop any database application using Java: `Driver`, `Connection`, `Statement`, and `ResultSet`. These interfaces define a framework for generic SQL database access. The JDBC API defines these interfaces, and the JDBC driver vendors provide the implementation for the interfaces. Programmers use these interfaces.

The relationship of these interfaces is shown in Figure 34.21. A JDBC application loads an appropriate driver using the `Driver` interface, connects to the database using the `Connection` interface, creates and executes SQL statements using the `Statement` interface, and processes the result using the `ResultSet` interface if the statements return results. Note some statements, such as SQL data definition statements and SQL data modification statements, do not return results.

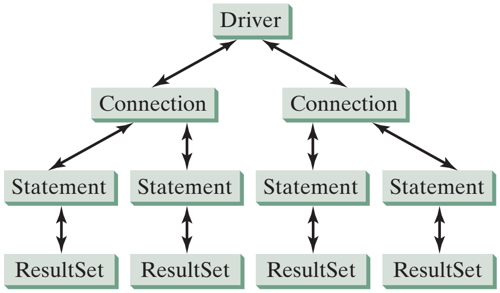


FIGURE 34.21 JDBC classes enable Java programs to connect to the database, send SQL statements, and process results.

The JDBC interfaces and classes are the building blocks in the development of Java database programs. A typical Java program takes the following steps to access a database.

1. Loading drivers.

An appropriate driver must be loaded using the statement shown below before connecting to a database.

```
Class.forName("JDBCDriverClass");
```

A driver is a concrete class that implements the `java.sql.Driver` interface. The drivers for MySQL, Oracle, and Java DB are listed in Table 34.3. If your program accesses several different databases, all their respective drivers must be loaded.

mysql-connector-java-5.1.26.jar
ojdbc6.jar

The most recent platform independent version of MySQL JDBC driver is **mysql-connector-java-5.1.26.jar.** This file is contained in a ZIP file downloadable from dev.mysql.com/downloads/connector/j/. The most recent version of Oracle JDBC driver is **ojdbc6.jar** (downloadable from www.oracle.com/technetwork/database/enterprise-edition/jdbc-112010-090769.html).

Table 34.3 JDBC Drivers

Database	*Driver Class*	*Source*
MySQL	`com.mysql.jdbc.Driver`	**mysql-connector-java-5.1.26.jar**
Oracle	`oracle.jdbc.driver.OracleDriver`	**ojdbc6.jar**
Java DB (embedded)	`org.apache.derby.jdbc.EmbeddedDriver`	**derby.jar**
Java DB (network)	`org.apache.derby.jdbc.ClientDriver`	**derbynet.jar**

Java DB has two versions: embedded and networked. Embedded version is used when you access Java DB locally, while the network version enables you to access Java DB on the network. To use these drivers, you have to add their jar files in the classpath using the following DOS command on Windows:

```
set classpath=%classpath%;c:\book\lib\mysql-connector-java-5.1.26.
jar;c:\book\lib\ojdbc6.jar;c:\program files\jdk1.8.0\db\lib\derby.
jar
```

If you use an IDE such as Eclipse or NetBeans, you need to add these jar files into the library in the IDE.

why load a driver?

Note

`com.mysql.jdbc.Driver` is a class in `mysql-connector-java-5.1.26.jar`, and `oracle.jdbc.driver.OracleDriver` is a class in `ojdbc6.jar`. `mysql-connector-java-5.1.26.jar`, `ojdbc6.jar`, and `derby.jar` contains many classes to support the driver. These classes are used by JDBC but not directly by JDBC programmers. When you use a class explicitly in the program, it is automatically loaded by the JVM. The driver classes, however, are not used explicitly in the program, so you have to write the code to tell the JVM to load them.

automatic driver discovery

Note

Java supports automatic driver discovery, so you don't have to load the driver explicitly. At the time of this writing, however, this feature is not supported for all database drivers. To be safe, load the driver explicitly.

2. Establishing connections.

To connect to a database, use the static method `getConnection(databaseURL)` in the `DriverManager` class, as follows:

```
Connection connection = DriverManager.getConnection(databaseURL);
```

where `databaseURL` is the unique identifier of the database on the Internet. Table 34.4 lists the URL patterns for the MySQL, Oracle, and Java DB.

Table 34.4 JDBC URLs

Database	*URL Pattern*
MySQL	`jdbc:mysql://hostname/dbname`
Oracle	`jdbc:oracle:thin:@hostname:port#:oracleDBSID`
Java DB (embedded)	`jdbc:derby:dbname`
Java DB (network)	`jdbc:derby://hostname/dbname`

The databaseURL for a MySQL database specifies the host name and database name to locate a database. For example, the following statement creates a Connection object for the local MySQL database javabook with username *scott* and password *tiger*:

connect MySQL DB

```
Connection connection = DriverManager.getConnection
  ("jdbc:mysql://localhost/javabook", "scott", "tiger");
```

Recall that by default, MySQL contains two databases named *mysql* and *test*. Section 34.3.2, Creating a Database, created a custom database named javabook. We will use javabook in the examples.

The databaseURL for an Oracle database specifies the *hostname*, the *port#* where the database listens for incoming connection requests, and the *oracleDBSID* database name to locate a database. For example, the following statement creates a Connection object for the Oracle database on liang.armstrong.edu with the username *scott* and password *tiger*:

connect Oracle DB

```
Connection connection = DriverManager.getConnection
  ("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl",
   "scott", "tiger");
```

3. Creating statements.

If a Connection object can be envisioned as a cable linking your program to a database, an object of Statement can be viewed as a cart that delivers SQL statements for execution by the database and brings the result back to the program. Once a Connection object is created, you can create statements for executing SQL statements as follows:

```
Statement statement = connection.createStatement();
```

4. Executing statements.

SQL data definition language (DDL) and update statements can be executed using executeUpdate(String sql), and an SQL query statement can be executed using executeQuery(String sql). The result of the query is returned in ResultSet. For example, the following code executes the SQL statement create table Temp (col1 char(5), col2 char(5)):

```
statement.executeUpdate
  ("create table Temp (col1 char(5), col2 char(5))");
```

This next code executes the SQL query select firstName, mi, lastName from Student where lastName = 'Smith':

```
// Select the columns from the Student table
ResultSet resultSet = statement.executeQuery
  ("select firstName, mi, lastName from Student where lastName "
    + " = 'Smith'");
```

5. Processing ResultSet.

The ResultSet maintains a table whose current row can be retrieved. The initial row position is null. You can use the next method to move to the next row and the various getter methods to retrieve values from a current row. For example, the following code displays all the results from the preceding SQL query:

```
// Iterate through the result and print the student names
while (resultSet.next())
  System.out.println(resultSet.getString(1) + " " +
    resultSet.getString(2) + " " + resultSet.getString(3));
```

The getString(1), getString(2), and getString(3) methods retrieve the column values for firstName, mi, and lastName, respectively. Alternatively, you can use getString("firstName"), getString("mi"), and getString("lastName") to retrieve the same three column values. The first execution of the next() method sets the current row to the first row in the result set, and subsequent invocations of the next() method set the current row to the second row, third row, and so on, to the last row.

Listing 34.1 is a complete example that demonstrates connecting to a database, executing a simple query, and processing the query result with JDBC. The program connects to a local MySQL database and displays the students whose last name is Smith.

Listing 34.1 SimpleJdbc.java

```
 1  import java.sql.*;
 2
 3  public class SimpleJdbc {
 4    public static void main(String[] args)
 5        throws SQLException, ClassNotFoundException {
 6      // Load the JDBC driver
 7      Class.forName("com.mysql.jdbc.Driver");
 8      System.out.println("Driver loaded");
 9
10      // Connect to a database
11      Connection connection = DriverManager.getConnection
12        ("jdbc:mysql://localhost/javabook", "scott", "tiger");
13      System.out.println("Database connected");
14
15      // Create a statement
16      Statement statement = connection.createStatement();
17
18      // Execute a statement
19      ResultSet resultSet = statement.executeQuery
20        ("select firstName, mi, lastName from Student where lastName "
21        + " = 'Smith'");
22
23      // Iterate through the result and print the student names
24      while (resultSet.next())
25        System.out.println(resultSet.getString(1) + "\t" +
26          resultSet.getString(2) + "\t" + resultSet.getString(3));
27
28      // Close the connection
29      connection.close();
30    }
31  }
```

load driver

connect database

create statement

execute statement

get result

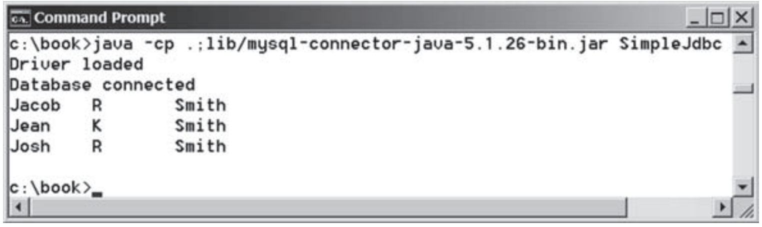
close connection

The statement in line 7 loads a JDBC driver for MySQL, and the statement in lines 11–13 connects to a local MySQL database. You can change them to connect to an Oracle or other databases. The program creates a Statement object (line 16), executes an SQL statement and returns a ResultSet object (lines 19–21), and retrieves the query result from the ResultSet object (lines 24–26). The last statement (line 29) closes the connection and releases resources related to the connection. You can rewrite this program using the try-with-resources syntax. See www.cs.armstrong.edu/liang/intro11e/html/SimpleJdbcWithAutoClose.html.

Note

If you run this program from the DOS prompt, specify the appropriate driver in the classpath, as shown in Figure 34.22.

run from DOS prompt



```
c:\book>java -cp .;lib/mysql-connector-java-5.1.26-bin.jar SimpleJdbc
Driver loaded
Database connected
Jacob   R       Smith
Jean    K       Smith
Josh    R       Smith

c:\book>_
```

FIGURE 34.22 You must include the driver file to run Java database programs.

The classpath directory and jar files are separated by commas. The period (`.`) represents the current directory. For convenience, the driver files are placed under the **lib** directory.

the semicolon issue

Caution

Do not use a semicolon (`;`) to end the Oracle SQL command in a Java program. The semicolon may not work with the Oracle JDBC drivers. It does work, however, with the other drivers used in this book.

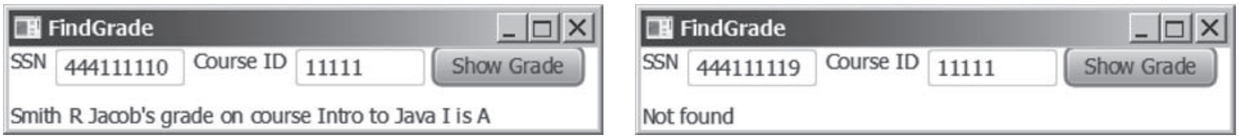
autocommit

Note

The `Connection` interface handles transactions and specifies how they are processed. By default, a new connection is in autocommit mode, and all its SQL statements are executed and committed as individual transactions. The commit occurs when the statement completes or the next execute occurs, whichever comes first. In the case of statements returning a result set, the statement completes when the last row of the result set has been retrieved or the result set has been closed. If a single statement returns multiple results, the commit occurs when all the results have been retrieved. You can use the `setAutoCommit(false)` method to disable autocommit, so all SQL statements are grouped into one transaction that is terminated by a call to either the `commit()` or the `rollback()` method. The `rollback()` method undoes all the changes made by the transaction.

34.4.2 Accessing a Database from JavaFX

This section gives an example that demonstrates connecting to a database from a JavaFX program. The program lets the user enter the SSN and the course ID to find a student's grade, as shown in Figure 34.23. The code in Listing 34.2 uses the MySQL database on the localhost.



SSN 444111110 Course ID 11111

Smith R Jacob's grade on course Intro to Java I is A

SSN 444111119 Course ID 11111

Not found

FIGURE 34.23 A JavaFX client can access the database on the server.

LISTING 34.2 `FindGrade.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.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import java.sql.*;
```

```

public class FindGrade extends Application {
  // Statement for executing queries
  private Statement stmt;
  private TextField tfSSN = new TextField();
  private TextField tfCourseId = new TextField();
  private Label lblStatus = new Label();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Initialize database connection and create a Statement object
    initializeDB();

    Button btShowGrade = new Button("Show Grade");
    HBox hBox = new HBox(5);
    hBox.getChildren().addAll(new Label("SSN"), tfSSN,
      new Label("Course ID"), tfCourseId, (btShowGrade));

    VBox vBox = new VBox(10);
    vBox.getChildren().addAll(hBox, lblStatus);

    tfSSN.setPrefColumnCount(6);
    tfCourseId.setPrefColumnCount(6);
    btShowGrade.setOnAction(e -> showGrade());

    // Create a scene and place it in the stage
    Scene scene = new Scene(vBox, 420, 80);
    primaryStage.setTitle("FindGrade"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  private void initializeDB() {
    try {
      // Load the JDBC driver
      Class.forName("com.mysql.jdbc.Driver");
//        Class.forName("oracle.jdbc.driver.OracleDriver");
      System.out.println("Driver loaded");

      // Establish a connection
      Connection connection = DriverManager.getConnection
        ("jdbc:mysql://localhost/javabook", "scott", "tiger");
//      ("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl",
//       "scott", "tiger");
      System.out.println("Database connected");

      // Create a statement
      stmt = connection.createStatement();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  private void showGrade() {
    String ssn = tfSSN.getText();
    String courseId = tfCourseId.getText();
    try {
      String queryString = "select firstName, mi, " +
        "lastName, title, grade from Student, Enrollment, Course " +
```

button listener

load driver

Oracle driver commented

connect to MySQL database

connect to Oracle commented

execute statement

show result

create statement

```
70          "where Student.ssn = '" + ssn + "' and Enrollment.courseId "
71          + "= '" + courseId +
72          "' and Enrollment.courseId = Course.courseId " +
73          " and Enrollment.ssn = Student.ssn";
74
75        ResultSet rset = stmt.executeQuery(queryString);
76
77        if (rset.next()) {
78          String lastName = rset.getString(1);
79          String mi = rset.getString(2);
80          String firstName = rset.getString(3);
81          String title = rset.getString(4);
82          String grade = rset.getString(5);
83
84          // Display result in a label
85          lblStatus.setText(firstName + " " + mi +
86            " " + lastName + "'s grade on course " + title + " is " +
87            grade);
88        } else {
89          lblStatus.setText("Not found");
90        }
91      }
92      catch (SQLException ex) {
93        ex.printStackTrace();
94      }
95    }
96  }
```

The `initializeDB()` method (lines 42–62) loads the MySQL driver (line 45), connects to the MySQL database on host `liang.armstrong.edu` (lines 50–55), and creates a statement (line 57).

security hole

Note

There is a *security hole* in this program. If you enter `1' or true or '1` in the `SSN` field, you will get the first student's score, because the query string now becomes

```
select firstName, mi, lastName, title, grade
from Student, Enrollment, Course
where Student.ssn = '1' or true or '1' and
      Enrollment.courseId = ' ' and
      Enrollment.courseId = Course.courseId and
      Enrollment.ssn = Student.ssn;
```

You can avoid this problem by using the `PreparedStatement` interface, which will be discussed in the next section.

Check Point

34.4.1 What are the advantages of developing database applications using Java?

34.4.2 Describe the following JDBC interfaces: `Driver`, `Connection`, `Statement`, and `ResultSet`.

34.4.3 How do you load a JDBC driver? What are the driver classes for MySQL, Oracle, and Java DB?

34.4.4 How do you create a database connection? What are the URLs for MySQL, Oracle, and Java DB?

34.4.5 How do you create a `Statement` and execute an SQL statement?

34.4.6 How do you retrieve values in a `ResultSet`?

34.4.7 Does JDBC automatically commit a transaction? How do you set autocommit to false?

34.5 PreparedStatement

Key Point

PreparedStatement enables you to create parameterized SQL statements.

Once a connection to a particular database is established, it can be used to send SQL statements from your program to the database. The Statement interface is used to execute static SQL statements that don't contain any parameters. The PreparedStatement interface, extending Statement, is used to execute a precompiled SQL statement with or without parameters. Since the SQL statements are precompiled, they are efficient for repeated executions.

A PreparedStatement object is created using the prepareStatement method in the Connection interface. For example, the following code creates a PreparedStatement for an SQL insert statement:

```
PreparedStatement preparedStatement = connection.prepareStatement
  ("insert into Student (firstName, mi, lastName) " +
   "values (?, ?, ?)");
```

This insert statement has three question marks as placeholders for parameters representing values for firstName, mi, and lastName in a record of the Student table.

As a subinterface of Statement, the PreparedStatement interface inherits all the methods defined in Statement. It also provides the methods for setting parameters in the object of PreparedStatement. These methods are used to set the values for the parameters before executing statements or procedures. In general, the setter methods have the following name and signature:

```
setX(int parameterIndex, X value);
```

where *X* is the type of the parameter, and parameterIndex is the index of the parameter in the statement. The index starts from 1. For example, the method setString(int parameterIndex, String value) sets a String value to the specified parameter.

The following statements pass the parameters "Jack", "A", and "Ryan" to the placeholders for firstName, mi, and lastName in preparedStatement:

```
preparedStatement.setString(1, "Jack");
preparedStatement.setString(2, "A");
preparedStatement.setString(3, "Ryan");
```

After setting the parameters, you can execute the prepared statement by invoking executeQuery() for a SELECT statement and executeUpdate() for a DDL or update statement.

The executeQuery() and executeUpdate() methods are similar to the ones defined in the Statement interface except that they don't have any parameters, because the SQL statements are already specified in the prepareStatement method when the object of PreparedStatement is created.

Using a prepared SQL statement, Listing 34.2 can be improved as in Listing 34.3.

Listing 34.3 FindGradeUsingPreparedStatement.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.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import java.sql.*;

```

```
public class FindGradeUsingPreparedStatement extends Application {
  // PreparedStatement for executing queries
  private PreparedStatement preparedStatement;
  private TextField tfSSN = new TextField();
  private TextField tfCourseId = new TextField();
  private Label lblStatus = new Label();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Initialize database connection and create a Statement object
    initializeDB();

    Button btShowGrade = new Button("Show Grade");
    HBox hBox = new HBox(5);
    hBox.getChildren().addAll(new Label("SSN"), tfSSN,
      new Label("Course ID"), tfCourseId, (btShowGrade));

    VBox vBox = new VBox(10);
    vBox.getChildren().addAll(hBox, lblStatus);

    tfSSN.setPrefColumnCount(6);
    tfCourseId.setPrefColumnCount(6);
    btShowGrade.setOnAction(e -> showGrade());

    // Create a scene and place it in the stage
    Scene scene = new Scene(vBox, 420, 80);
    primaryStage.setTitle("FindGrade"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  private void initializeDB() {
    try {
      // Load the JDBC driver
      Class.forName("com.mysql.jdbc.Driver");
//        Class.forName("oracle.jdbc.driver.OracleDriver");
      System.out.println("Driver loaded");

      // Establish a connection
      Connection connection = DriverManager.getConnection
        ("jdbc:mysql://localhost/javabook", "scott", "tiger");
//      ("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl",
//       "scott", "tiger");
      System.out.println("Database connected");

      String queryString = "select firstName, mi, " +
        "lastName, title, grade from Student, Enrollment, Course " +
        "where Student.ssn = ? and Enrollment.courseId = ? " +
        "and Enrollment.courseId = Course.courseId";

      // Create a statement
      preparedStatement = connection.prepareStatement(queryString);
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  private void showGrade() {
    String ssn = tfSSN.getText();
```

prepare statement

load driver

connect database

placeholder

```
71        String courseId = tfCourseId.getText();
72        try {
73          preparedStatement.setString(1, ssn);
74          preparedStatement.setString(2, courseId);
75          ResultSet rset = preparedStatement.executeQuery();
76
77          if (rset.next()) {
78            String lastName = rset.getString(1);
79            String mi = rset.getString(2);
80            String firstName = rset.getString(3);
81            String title = rset.getString(4);
82            String grade = rset.getString(5);
83
84            // Display result in a label
85            lblStatus.setText(firstName + " " + mi +
86              " " + lastName + "'s grade on course " + title + " is " +
87              grade);
88          } else {
89            lblStatus.setText("Not found");
90          }
91        }
92        catch (SQLException ex) {
93          ex.printStackTrace();
94        }
95      }
96    }
```

execute statement

show result

This example does exactly the same thing as Listing 34.2 except that it uses the prepared statement to dynamically set the parameters. The code in this example is almost the same as in the preceding example. The new code is highlighted.

A prepared query string is defined in lines 56–59 with `ssn` and `courseId` as parameters. An SQL prepared statement is obtained in line 62. Before executing the query, the actual values of `ssn` and `courseId` are set to the parameters in lines 73–74. Line 75 executes the prepared statement.

Check Point

34.5.1 Describe prepared statements. How do you create instances of `PreparedStatement`? How do you execute a `PreparedStatement`? How do you set parameter values in a `PreparedStatement`?

34.5.2 What are the benefits of using prepared statements?

34.6 CallableStatement

Key Point

`CallableStatement` enables you to execute SQL stored procedures.

The `CallableStatement` interface is designed to execute SQL-stored procedures. The procedures may have `IN`, `OUT`, or `IN OUT` parameters. An `IN` parameter receives a value passed to the procedure when it is called. An `OUT` parameter returns a value after the procedure is completed, but it doesn't contain any value when the procedure is called. An `IN OUT` parameter contains a value passed to the procedure when it is called and returns a value after it is completed. For example, the following procedure in Oracle PL/SQL has `IN` parameter `p1`, `OUT` parameter `p2`, and `IN OUT` parameter `p3`:

IN parameter
OUT parameter
IN OUT parameter

```
create or replace procedure sampleProcedure
  (p1 in varchar, p2 out number, p3 in out integer) is
begin
  /* do something */
end sampleProcedure;
/
```

Note

The syntax of stored procedures is vendor specific. We use both Oracle and MySQL for demonstrations of stored procedures in this book.

A CallableStatement object can be created using the prepareCall(String call) method in the Connection interface. For example, the following code creates a CallableStatement cstmt on Connection connection for the procedure sampleProcedure:

```
CallableStatement callableStatement = connection.prepareCall(
  "{call sampleProcedure(?, ?, ?)}");
```

{call sampleProcedure(?, ?, ...)} is referred to as the *SQL escape syntax*, which signals the driver that the code within it should be handled differently. The driver parses the escape syntax and translates it into code that the database understands. In this example, sampleProcedure is an Oracle procedure. The call is translated to the string begin sampleProcedure(?, ?, ?); end and passed to an Oracle database for execution.

You can call procedures as well as functions. The syntax to create an SQL callable statement for a function is:

```
{? = call functionName(?, ?, ...)}
```

CallableStatement inherits PreparedStatement. Additionally, the CallableStatement interface provides methods for registering the OUT parameters and for getting values from the OUT parameters.

Before calling an SQL procedure, you need to use appropriate setter methods to pass values to IN and IN OUT parameters, and use registerOutParameter to register OUT and IN OUT parameters. For example, before calling procedure sampleProcedure, the following statements pass values to parameters p1 (IN) and p3 (IN OUT) and register parameters p2 (OUT) and p3 (IN OUT):

```
callableStatement.setString(1, "Dallas"); // Set Dallas to p1
callableStatement.setLong(3, 1); // Set 1 to p3
// Register OUT parameters
callableStatement.registerOutParameter(2, java.sql.Types.DOUBLE);
callableStatement.registerOutParameter(3, java.sql.Types.INTEGER);
```

You can use execute() or executeUpdate() to execute the procedure depending on the type of SQL statement, then use getter methods to retrieve values from the OUT parameters. For example, the next statements retrieve the values from parameters p2 and p3:

```
double d = callableStatement.getDouble(2);
int i = callableStatement.getInt(3);
```

Let us define a MySQL function that returns the number of the records in the table that match the specified firstName and lastName in the Student table.

```
/* For the callable statement example. Use MySQL version 5 */
drop function if exists studentFound;

delimiter //

create function studentFound(first varchar(20), last varchar(20))
  returns int
begin
  declare result int;

  select count(*) into result
  from Student
  where Student.firstName = first and
    Student.lastName = last;

  return result;
end;
//

delimiter ;
/* Please note that there is a space between delimiter and ; */
```

If you use an Oracle database, the function can be defined as follows:

```
create or replace function studentFound
  (first varchar2, last varchar2)
  /* Do not name firstName and lastName. */
  return number is numberOfSelectedRows number := 0;
begin
  select count(*) into numberOfSelectedRows
  from Student
  where Student.firstName = first and
    Student.lastName = last;

  return numberOfSelectedRows;
end studentFound;
/
```

Suppose the function studentFound is already created in the database. Listing 34.4 gives an example that tests this function using callable statements.

Listing 34.4 TestCallableStatement.java

```
import java.sql.*;

public class TestCallableStatement {
  /** Creates new form TestTableEditor */
  public static void main(String[] args) throws Exception {
    Class.forName("com.mysql.jdbc.Driver");
    Connection connection = DriverManager.getConnection(
      "jdbc:mysql://localhost/javabook",
      "scott", "tiger");
//    Connection connection = DriverManager.getConnection(
//      ("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl",
//      "scott", "tiger");

    // Create a callable statement
    CallableStatement callableStatement = connection.prepareCall(
      "{? = call studentFound(?, ?)}");

    java.util.Scanner input = new java.util.Scanner(System.in);
    System.out.print("Enter student's first name: ");
    String firstName = input.nextLine();
    System.out.print("Enter student's last name: ");
    String lastName = input.nextLine();

    callableStatement.setString(2, firstName);
    callableStatement.setString(3, lastName);
    callableStatement.registerOutParameter(1, Types.INTEGER);
    callableStatement.execute();
```

load driver
connect database
create callable statement
enter firstName
enter lastName
set IN parameter
set IN parameter
register OUT parameter
execute statement

get OUT parameter

```
28
29      if (callableStatement.getInt(1) >= 1)
30        System.out.println(firstName + " " + lastName +
31          " is in the database");
32      else
33        System.out.println(firstName + " " + lastName +
34          " is not in the database");
35    }
36  }
```

```
Enter student's first name: Jacob ↵Enter
Enter student's last name: Smith ↵Enter
Jacob Smith is in the database
```

```
Enter student's first name: John ↵Enter
Enter student's last name: Smith ↵Enter
John Smith is not in the database
```

The program loads a MySQL driver (line 6), connects to a MySQL database (lines 7–9), and creates a callable statement for executing the function `studentFound` (lines 15–16).

The function's first parameter is the return value; its second and third parameters correspond to the first and last names. Before executing the callable statement, the program sets the first name and last name (lines 24–25) and registers the `OUT` parameter (line 26). The statement is executed in line 27.

The function's return value is obtained in line 29. If the value is greater than or equal to `1`, the student with the specified first and last name is found in the table.

Check Point

34.6.1 Describe callable statements. How do you create instances of `CallableStatement`? How do you execute a `CallableStatement`? How do you register `OUT` parameters in a `CallableStatement`?

34.7 Retrieving Metadata

Key Point

The database metadata such as database URL, username, and JDBC driver name can be obtained using the `DatabaseMetaData` *interface and result set metadata such as table column count and column names can be obtained using the* `ResultSetMetaData` *interface.*

database metadata

JDBC provides the `DatabaseMetaData` interface for obtaining database-wide information, and the `ResultSetMetaData` interface for obtaining information on a specific `ResultSet`.

34.7.1 Database Metadata

The `Connection` interface establishes a connection to a database. It is within the context of a connection that SQL statements are executed and results are returned. A connection also provides access to database metadata information that describes the capabilities of the database, supported SQL grammar, stored procedures, and so on. To obtain an instance of `DatabaseMetaData` for a database, use the `getMetaData` method on a `Connection` object like this:

```
DatabaseMetaData dbMetaData = connection.getMetaData();
```

If your program connects to a local MySQL database, the program in Listing 34.5 displays the database information statements shown in Figure 34.24.

LISTING 34.5 TestDatabaseMetaData.java

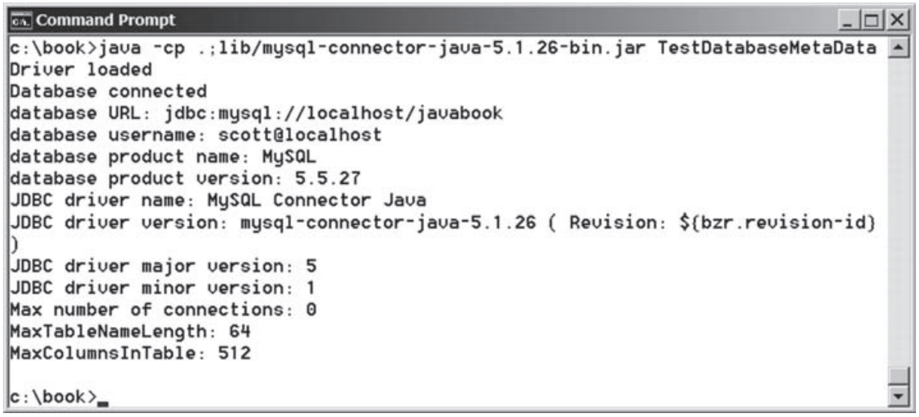
```java
import java.sql.*;

public class TestDatabaseMetaData {
  public static void main(String[] args)
      throws SQLException, ClassNotFoundException {
    // Load the JDBC driver
    Class.forName("com.mysql.jdbc.Driver");
    System.out.println("Driver loaded");

    // Connect to a database
    Connection connection = DriverManager.getConnection
      ("jdbc:mysql://localhost/javabook", "scott", "tiger");
    System.out.println("Database connected");

    DatabaseMetaData dbMetaData = connection.getMetaData();
    System.out.println("database URL: " + dbMetaData.getURL());
    System.out.println("database username: " +
      dbMetaData.getUserName());
    System.out.println("database product name: " +
      dbMetaData.getDatabaseProductName());
    System.out.println("database product version: " +
      dbMetaData.getDatabaseProductVersion());
    System.out.println("JDBC driver name: " +
      dbMetaData.getDriverName());
    System.out.println("JDBC driver version: " +
      dbMetaData.getDriverVersion());
    System.out.println("JDBC driver major version: " +
      dbMetaData.getDriverMajorVersion());
    System.out.println("JDBC driver minor version: " +
      dbMetaData.getDriverMinorVersion());
    System.out.println("Max number of connections: " +
      dbMetaData.getMaxConnections());
    System.out.println("MaxTableNameLength: " +
      dbMetaData.getMaxTableNameLength());
    System.out.println("MaxColumnsInTable: " +
      dbMetaData.getMaxColumnsInTable());

    // Close the connection
    connection.close();
  }
}
```

load driver

connect database

database metadata

get metadata



```
c:\book>java -cp .;lib/mysql-connector-java-5.1.26-bin.jar TestDatabaseMetaData
Driver loaded
Database connected
database URL: jdbc:mysql://localhost/javabook
database username: scott@localhost
database product name: MySQL
database product version: 5.5.27
JDBC driver name: MySQL Connector Java
JDBC driver version: mysql-connector-java-5.1.26 ( Revision: ${bzr.revision-id}
)
JDBC driver major version: 5
JDBC driver minor version: 1
Max number of connections: 0
MaxTableNameLength: 64
MaxColumnsInTable: 512

c:\book>_
```

FIGURE 34.24 The DatabaseMetaData interface enables you to obtain database information.

34.7.2 Obtaining Database Tables

You can identify the tables in the database through database metadata using the **getTables** method. Listing 34.6 displays all the user tables in the javabook database on a local MySQL database. Figure 34.25 shows a sample output of the program.

Listing 34.6 FindUserTables.java

load driver

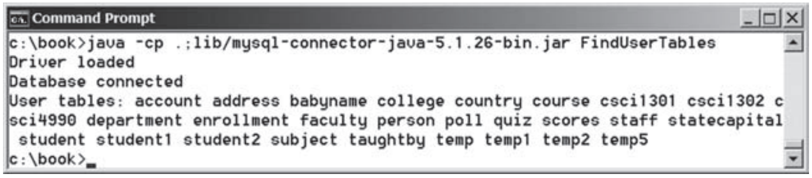
connect database

database metadata

obtain tables

get table names

```
 1  import java.sql.*;
 2
 3  public class FindUserTables {
 4    public static void main(String[] args)
 5        throws SQLException, ClassNotFoundException {
 6      // Load the JDBC driver
 7      Class.forName("com.mysql.jdbc.Driver");
 8      System.out.println("Driver loaded");
 9
10      // Connect to a database
11      Connection connection = DriverManager.getConnection
12        ("jdbc:mysql://localhost/javabook", "scott", "tiger");
13      System.out.println("Database connected");
14
15      DatabaseMetaData dbMetaData = connection.getMetaData();
16
17      ResultSet rsTables = dbMetaData.getTables(null, null, null,
18        new String[] {"TABLE"});
19      System.out.print("User tables: ");
20      while (rsTables.next())
21        System.out.print(rsTables.getString("TABLE_NAME") + " ");
22
23      // Close the connection
24      connection.close();
25    }
26  }
```



```
c:\book>java -cp .;lib/mysql-connector-java-5.1.26-bin.jar FindUserTables
Driver loaded
Database connected
User tables: account address babyname college country course csci1301 csci1302 c
sci4990 department enrollment faculty person poll quiz scores staff statecapital
 student student1 student2 subject taughtby temp temp1 temp2 temp5
c:\book>_
```

Figure 34.25 You can find all the tables in the database.

Line 17 obtains table information in a result set using the **getTables** method. One of the columns in the result set is TABLE_NAME. Line 21 retrieves the table name from this result set column.

34.7.3 Result Set Metadata

The **ResultSetMetaData** interface describes information pertaining to the result set. A **ResultSetMetaData** object can be used to find the types and properties of the columns in a **ResultSet**. To obtain an instance of **ResultSetMetaData**, use the **getMetaData** method on a result set like this:

```
ResultSetMetaData rsMetaData = resultSet.getMetaData();
```

You can use the getColumnCount() method to find the number of columns in the result and the getColumnName(int) method to get the column names. For example, Listing 34.7 displays all the column names and contents resulting from the SQL SELECT statement *select * from Enrollment*. The output is shown in Figure 34.26.

LISTING 34.7 TestResultSetMetaData.java

```java
import java.sql.*;

public class TestResultSetMetaData {
  public static void main(String[] args)
      throws SQLException, ClassNotFoundException {
    // Load the JDBC driver
    Class.forName("com.mysql.jdbc.Driver");
    System.out.println("Driver loaded");

    // Connect to a database
    Connection connection = DriverManager.getConnection
      ("jdbc:mysql://localhost/javabook", "scott", "tiger");
    System.out.println("Database connected");

    // Create a statement
    Statement statement = connection.createStatement();

    // Execute a statement
    ResultSet resultSet = statement.executeQuery
      ("select * from Enrollment");

    ResultSetMetaData rsMetaData = resultSet.getMetaData();
    for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
      System.out.printf("%-12s\t", rsMetaData.getColumnName(i));
    System.out.println();

    // Iterate through the result and print the students' names
    while (resultSet.next()) {
      for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
        System.out.printf("%-12s\t", resultSet.getObject(i));
      System.out.println();
    }

    // Close the connection
    connection.close();
  }
}
```

load driver

connect database

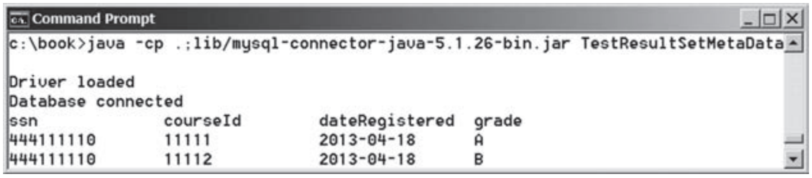
create statement

create result set

result set metadata

column count

column name



```
c:\book>java -cp .;lib/mysql-connector-java-5.1.26-bin.jar TestResultSetMetaData

Driver loaded
Database connected
ssn             courseId        dateRegistered  grade
444111110       11111           2013-04-18      A
444111110       11112           2013-04-18      B
```

FIGURE 34.26 The ResultSetMetaData interface enables you to obtain result set information.

Check Point

34.7.1 What is DatabaseMetaData for? Describe the methods in DatabaseMetaData. How do you get an instance of DatabaseMetaData?

34.7.2 What is ResultSetMetaData for? Describe the methods in ResultSetMetaData. How do you get an instance of ResultSetMetaData?

34.7.3 How do you find the number of columns in a result set? How do you find the column names in a result set?

Key Terms

candidate key	34-5	integrity constraint	34-4
database system	34-2	primary key	34-5
domain constraint	34-5	relational database	34-5
foreign key	34-5	Structured Query Language (SQL)	34-6
foreign key constraint	34-5	superkey	34-5

Chapter Summary

1. This chapter introduced the concepts of *database systems*, *relational databases*, *relational data models*, *data integrity*, and *SQL*. You learned how to develop database applications using Java.

2. The Java API for developing Java database applications is called *JDBC*. JDBC provides Java programmers with a uniform interface for accessing and manipulating relational databases.

3. The JDBC API consists of classes and interfaces for establishing connections with databases, sending SQL statements to databases, processing the results of SQL statements, and obtaining database metadata.

4. Since a JDBC driver serves as the interface to facilitate communications between JDBC and a proprietary database, JDBC drivers are database specific. If you use a driver, make sure it is in the classpath before running the program.

5. Four key interfaces are needed to develop any database application using Java: Driver, Connection, Statement, and ResultSet. These interfaces define a framework for generic SQL database access. The JDBC driver vendors provide implementation for them.

6. A JDBC application loads an appropriate driver using the Driver interface, connects to the database using the Connection interface, creates and executes SQL statements using the Statement interface, and processes the result using the ResultSet interface if the statements return results.

7. The PreparedStatement interface is designed to execute dynamic SQL statements with parameters. These SQL statements are precompiled for efficient use when repeatedly executed.

8. Database *metadata* is information that describes the database itself. JDBC provides the DatabaseMetaData interface for obtaining database-wide information and the ResultSetMetaData interface for obtaining information on the specific ResultSet.

Quiz

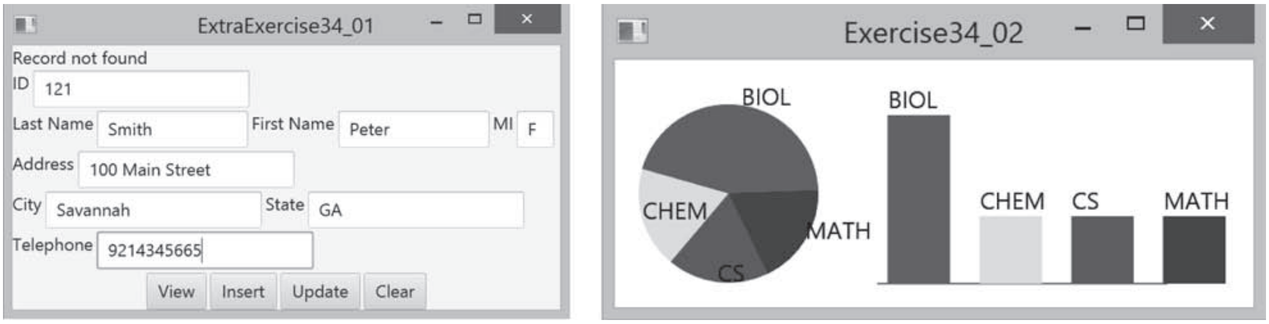
Answer the quiz for this chapter online at the book Companion Website.

Programming Exercises

MyProgrammingLab™

***34.1** (*Access and update a* `Staff` *table*) Write a program that views, inserts, and updates staff information stored in a database, as shown in Figure 34.27a. The *View* button displays a record with a specified ID. The *Insert* button inserts a new record. The *Update* button updates the record for the specified ID. The `Staff` table is created as follows:

```
create table Staff (
  id char(9) not null,
  lastName varchar(15),
  firstName varchar(15),
  mi char(1),
  address varchar(20),
  city varchar(20),
  state char(2),
  telephone char(10),
  email varchar(40),
  primary key (id)
);
```


ExtraExercise34_01

Record not found

ID: 121

Last Name: Smith First Name: Peter MI: F

Address: 100 Main Street

City: Savannah State: GA

Telephone: 9214345665

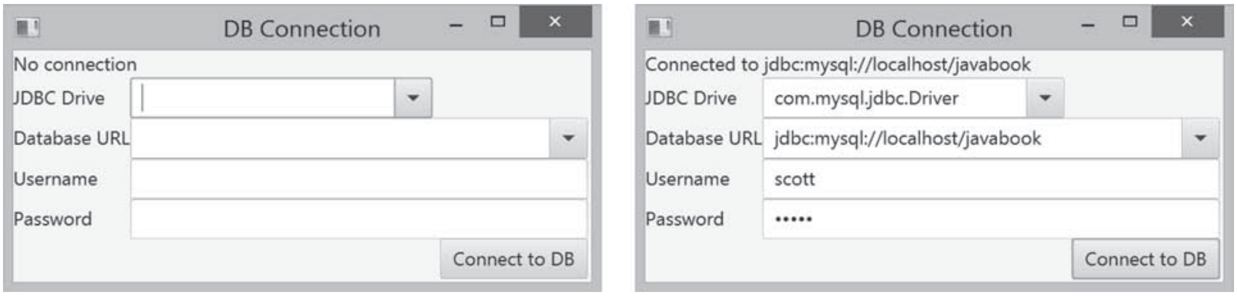
(a) (b)

Figure 34.27 (a) The program lets you view, insert, and update staff information. (b) The `PieChart` and `BarChart` components display the query data obtained from the data module.

****34.2** (*Visualize data*) Write a program that displays the number of students in each department in a pie chart and a bar chart, as shown in Figure 34.27b. The `PieChart` and `BarChart` classes are created in Programming Exercises 14.12 and 14.13. The number of students for each department can be obtained from the `Student` table (see Figure 34.4) using the following SQL statement:

```
select deptId, count(*)
from Student
where deptId is not null
group by deptId;
```

***34.3** (*Connection dialog*) Develop a subclass of `BorderPane` named `DBConnectionPane` that enables the user to select or enter a JDBC driver and a URL and to enter a username and password, as shown in Figure 34.28. When the *Connect to DB* button is clicked, a `Connection` object for the database is stored in the `connection` property. You can then use the `getConnection()` method to return the connection.



DB Connection

No connection

JDBC Drive

Database URL

Username

Password

Connect to DB

DB Connection

Connected to jdbc:mysql://localhost/javabook

JDBC Drive com.mysql.jdbc.Driver

Database URL jdbc:mysql://localhost/javabook

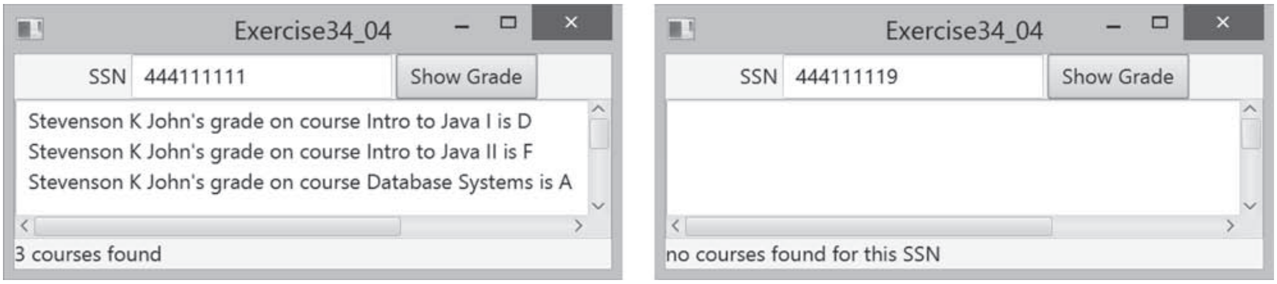
Username scott

Password •••••

Connect to DB

Figure 34.28 The `DBConnectionPane` component enables the user to enter database information.

***34.4** (*Find grades*) Listing 34.2, FindGrade.java, presented a program that finds a student's grade for a specified course. Rewrite the program to find all the grades for a specified student, as shown in Figure 34.29.



Exercise34_04

SSN 444111111 Show Grade

```
Stevenson K John's grade on course Intro to Java I is D
Stevenson K John's grade on course Intro to Java II is F
Stevenson K John's grade on course Database Systems is A
```

3 courses found

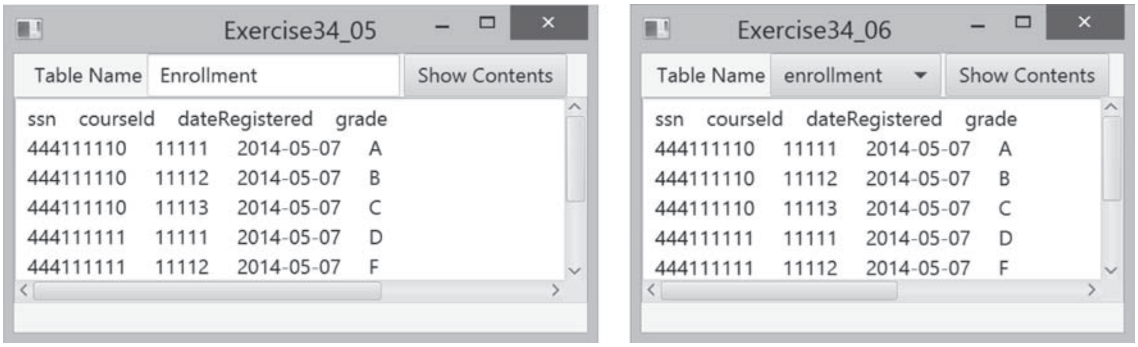
Exercise34_04

SSN 444111119 Show Grade

no courses found for this SSN

Figure 34.29 The program displays the grades for the courses for a specified student.

***34.5** (*Display table contents*) Write a program that displays the content for a given table. As shown in Figure 34.30a, you enter a table and click the *Show Contents* button to display the table contents in the text area.



Exercise34_05

Table Name Enrollment Show Contents

```
ssn    courseId  dateRegistered  grade
444111110  11111  2014-05-07  A
444111110  11112  2014-05-07  B
444111110  11113  2014-05-07  C
444111111  11111  2014-05-07  D
444111111  11112  2014-05-07  F
```

(a)

Exercise34_06

Table Name enrollment Show Contents

```
ssn    courseId  dateRegistered  grade
444111110  11111  2014-05-07  A
444111110  11112  2014-05-07  B
444111110  11113  2014-05-07  C
444111111  11111  2014-05-07  D
444111111  11112  2014-05-07  F
```

(b)

Figure 34.30 (a) Enter a table name to display the table contents. (b) Select a table name from the combo box to display its contents.

***34.6** (*Find tables and showing their contents*) Write a program that fills in table names in a combo box, as shown in Figure 34.30b. You can select a table from the combo box to display its contents in the text area.

****34.7** (*Populate Quiz table*) Create a table named `Quiz` as follows:

```
create table Quiz(
  questionId int,
  question varchar(4000),
  choicea varchar(1000),
  choiceb varchar(1000),
  choicec varchar(1000),
  choiced varchar(1000),
  answer varchar(5));
```

The **Quiz** table stores multiple-choice questions. Suppose the multiple-choice questions are stored in a text file accessible from http://www.cs.armstrong.edu/liang/data/Quiz.txt in the following format:

```
1. question1
a. choice a
b. choice b
c. choice c
d. choice d
Answer:cd

2. question2
a. choice a
b. choice b
c. choice c
d. choice d
Answer:a

...
```

Write a program that reads the data from the file and populate it into the **Quiz** table.

***34.8** (*Populate Salary table*) Create a table named **Salary** as follows:

```
create table Salary(
  firstName varchar(100),
  lastName varchar(100),
  rank varchar(15),
  salary float);
```

Obtain the data for salary from http://cs.armstrong.edu/liang/data/Salary.txt and populate it into the **Salary** table in the database.

***34.9** (*Copy table*) Suppose the database contains a student table defined as follows:

```
create table Student1 (
  username varchar(50) not null,
  password varchar(50) not null,
  fullname varchar(200) not null,
  constraint pkStudent primary key (username)
);
```

Create a new table named **Student2** as follows:

```
create table Student2 (
  username varchar(50) not null,
  password varchar(50) not null,
  firstname varchar(100),
  lastname varchar(100),
  constraint pkStudent primary key (username)
);
```

A full name is in the form of **firstname mi lastname** or **firstname lastname**. For example, **John K Smith** is a full name. Write a program that copies table Student1 into Student2. Your task is to split a full name into firstname, mi, and lastname for each record in Student1 and store a new record into Student2.

***34.10** (*Record unsubmitted exercises*) The following three tables store information on students, assigned exercises, and exercise submission in LiveLab. LiveLab is an automatic grading system for grading programming exercises.

```
create table AGSStudent (
  username varchar(50) not null,
  password varchar(50) not null,
  fullname varchar(200) not null,
  instructorEmail varchar(100) not null,
  constraint pkAGSStudent primary key (username)
);

 create table ExerciseAssigned (
  instructorEmail varchar(100),
  exerciseName varchar(100),
  maxscore double default 10,
  constraint pkCustomExercise primary key
    (instructorEmail, exerciseName)
);

create table AGSLog (
  username varchar(50), /* This is the student's user name */
  exerciseName varchar(100), /* This is the exercise */
  score double default null,
  submitted bit default 0,
  constraint pkLog primary key (username, exerciseName)
);
```

The AGSStudent table stores the student information. The ExerciseAssigned table assigns the exercises by an instructor. The AGSLog table stores the grading results. When a student submits an exercise, a record is stored in the AGSLog table. However, there is no record in AGSLog if a student did not submit the exercise.

Write a program that adds a new record for each student and an assigned exercise to the student in the AGSLog table if a student has not submitted the exercise. The record should have 0 on score and submitted. For example, if the tables contain the following data in AGSLog before you run this program, the AGSLog table now contains the new records after the program runs.

AGSStudent

username	password	fullname	instructorEmail
abc	p1	John Roo	t@gmail.com
cde	p2	Yao Mi	c@gmail.com
wbc	p3	F3	t@gmail.com

ExerciseAssigned

instructorEmail	exerciseName	maxScore
t@gmail.com	e1	10
t@gmail.com	e2	10
c@gmail.com	e1	4
c@gmail.com	e4	20

AGSLog

username	exerciseName	score	submitted
abc	e1	9	1
wbc	e2	7	1

AGSLog after the program runs

username	exerciseName	score	submitted
abc	e1	9	1
wbc	e2	7	1
abc	e2		0
wbc	e1		0
cde	e1		0
cde	e4		0

***34.11** (*Baby names*) Create the following table:

```
create table Babyname (
  year integer,
  name varchar(50),
  gender char(1),
  count integer,
  constraint pkBabyname primary key (year, name, gender)
);
```

The baby name ranking data was described in Programming Exercise 12.31. Write a program to read data from the following URL and store into the Babyname table. https://liveexample.pearsoncmg.com/data/babynamesranking2001.txt,

. . .

https://liveexample.pearsoncmg.com/data/babynamesranking2010.txt.

CHAPTER

35

Advanced Java Database Programming

Objectives

- To create a universal SQL client for accessing local or remote database (§35.2).
- To execute SQL statements in a batch mode (§35.3).
- To process updatable and scrollable result sets (§35.4).
- To simplify Java database programming using `RowSet` (§35.5).
- To store and retrieve images in JDBC (§35.6).

35.1 Introduction

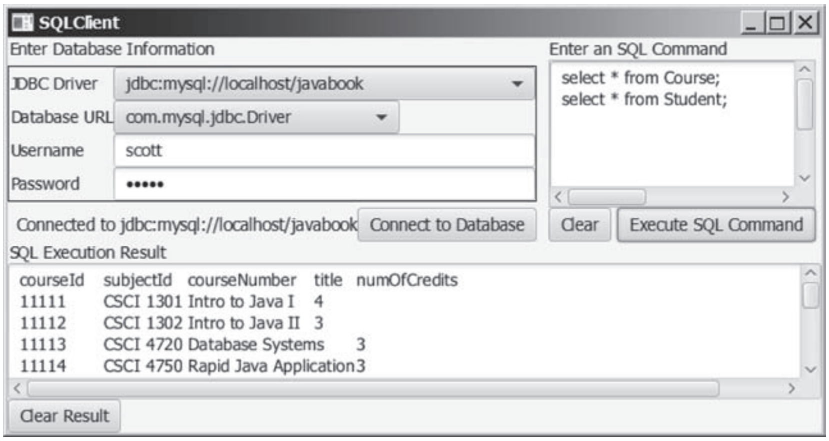
This chapter introduces advanced features for Java database programming.

Chapter 34 introduced JDBC's basic features. This chapter covers its advanced features. You will learn how to develop a universal SQL client for accessing any local or remote relational database, learn how to execute statements in a batch mode to improve performance, learn scrollable result sets and how to update a database through result sets, learn how to use `RowSet` to simplify database access, and learn how to store and retrieve images.

35.2 A Universal SQL Client

This section develops a universal SQL client for connecting and accessing any SQL database.

In Chapter 34, you used various drivers to connect to the database, created statements for executing SQL statements, and processed the results from SQL queries. This section presents a universal SQL client that enables you to connect to any relational database and execute SQL commands interactively, as shown in Figure 35.1. The client can connect to any JDBC data source and can submit SQL SELECT commands and non-SELECT commands for execution. The execution result is displayed for the SELECT queries, and the execution status is displayed for the non-SELECT commands. Listing 35.1 gives the program.



JDBC Driver: jdbc:mysql://localhost/javabook
Database URL: com.mysql.jdbc.Driver
Username: scott
Password: •••••

```
select * from Course;
select * from Student;
```

Connected to jdbc:mysql://localhost/javabook

```
courseId  subjectId  courseNumber  title  numOfCredits
11111     CSCI 1301 Intro to Java I   4
11112     CSCI 1302 Intro to Java II  3
11113     CSCI 4720 Database Systems     3
11114     CSCI 4750 Rapid Java Application3
```

FIGURE 35.1 You can connect to any JDBC data source and execute SQL commands interactively.

LISTING 35.1 SQLClient.java

```
import java.sql.*;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.control.PasswordField;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class SQLClient extends Application {
  // Connection to the database
  private Connection connection;

  // Statement to execute SQL commands
  private Statement statement;

  // Text area to enter SQL commands
  private TextArea tasqlCommand = new TextArea();

  // Text area to display results from SQL commands
  private TextArea taSQLResult = new TextArea();

  // DBC info for a database connection
  private TextField tfUsername = new TextField();
  private PasswordField pfPassword = new PasswordField();
  private ComboBox<String> cboURL = new ComboBox<>();
  private ComboBox<String> cboDriver = new ComboBox<>();

  private Button btExecuteSQL = new Button("Execute SQL Command");
  private Button btClearSQLCommand = new Button("Clear");
  private Button btConnectDB = new Button("Connect to Database");
  private Button btClearSQLResult = new Button("Clear Result");
  private Label lblConnectionStatus
    = new Label("No connection now");

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    cboURL.getItems().addAll(FXCollections.observableArrayList(
      "jdbc:mysql://localhost/javabook",
      "jdbc:mysql://liang.armstrong.edu/javabook",
      "jdbc:odbc:exampleMDBDataSource",
      "jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl"));
    cboURL.getSelectionModel().selectFirst();

    cboDriver.getItems().addAll(FXCollections.observableArrayList(
     "com.mysql.jdbc.Driver", "sun.jdbc.odbc.dbcOdbcDriver",
     "oracle.jdbc.driver.OracleDriver"));
    cboDriver.getSelectionModel().selectFirst();

    // Create UI for connecting to the database
    GridPane gridPane = new GridPane();
    gridPane.add(cboURL, 1, 0);
    gridPane.add(cboDriver, 1, 1);
    gridPane.add(tfUsername, 1, 2);
    gridPane.add(pfPassword, 1, 3);
    gridPane.add(new Label("JDBC Driver"), 0, 0);
    gridPane.add(new Label("Database URL"), 0, 1);
    gridPane.add(new Label("Username"), 0, 2);
    gridPane.add(new Label("Password"), 0, 3);

    HBox hBoxConnection = new HBox();
    hBoxConnection.getChildren().addAll(
      lblConnectionStatus, btConnectDB);
    hBoxConnection.setAlignment(Pos.CENTER_RIGHT);

    VBox vBoxConnection = new VBox(5);
    vBoxConnection.getChildren().addAll(
      new Label("Enter Database Information"),
      gridPane, hBoxConnection);

    gridPane.setStyle("-fx-border-color: black;");

    HBox hBoxSQLCommand = new HBox(5);
    hBoxSQLCommand.getChildren().addAll(
      btClearSQLCommand, btExecuteSQL);
    hBoxSQLCommand.setAlignment(Pos.CENTER_RIGHT);

    BorderPane borderPaneSqlCommand = new BorderPane();
    borderPaneSqlCommand.setTop(
      new Label("Enter an SQL Command"));
    borderPaneSqlCommand.setCenter(
      new ScrollPane(tasqlCommand));
    borderPaneSqlCommand.setBottom(
      hBoxSQLCommand);

    HBox hBoxConnectionCommand = new HBox(10);
    hBoxConnectionCommand.getChildren().addAll(
      vBoxConnection, borderPaneSqlCommand);

    BorderPane borderPaneExecutionResult = new BorderPane();
    borderPaneExecutionResult.setTop(
      new Label("SQL Execution Result"));
    borderPaneExecutionResult.setCenter(taSQLResult);
    borderPaneExecutionResult.setBottom(btClearSQLResult);

    BorderPane borderPane = new BorderPane();
    borderPane.setTop(hBoxConnectionCommand);
    borderPane.setCenter(borderPaneExecutionResult);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 670, 400);
    primaryStage.setTitle("SQLClient"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    btConnectDB.setOnAction(e -> connectToDB());
    btExecuteSQL.setOnAction(e -> executeSQL());
    btClearSQLCommand.setOnAction(e -> tasqlCommand.setText(null));
    btClearSQLResult.setOnAction(e -> taSQLResult.setText(null));
  }

  /** Connect to DB */
  private void connectToDB() {
    // Get database information from the user input
    String driver = cboDriver
      .getSelectionModel().getSelectedItem();
    String url = cboURL.getSelectionModel().getSelectedItem();
    String username = tfUsername.getText().trim();
    String password = pfPassword.getText().trim();

    // Connection to the database
    try {
      Class.forName(driver);
      connection = DriverManager.getConnection(
        url, username, password);
      lblConnectionStatus.setText("Connected to " + url);
    }
    catch (java.lang.Exception ex) {
      ex.printStackTrace();
    }
  }

  /** Execute SQL commands */
  private void executeSQL() {
    if (connection == null) {
      taSQLResult.setText("Please connect to a database first");
      return;
    }
    else {
      String sqlCommands = tasqlCommand.getText().trim();
      String[] commands = sqlCommands.replace('\n', ' ').split(";");

      for (String aCommand: commands) {
        if (aCommand.trim().toUpperCase().startsWith("SELECT")) {
          processSQLSelect(aCommand);
        }
        else {
          processSQLNonSelect(aCommand);
        }
      }
    }
  }

  /** Execute SQL SELECT commands */
  private void processSQLSelect(String sqlCommand) {
    try {
      // Get a new statement for the current connection
      statement = connection.createStatement();

      // Execute a SELECT SQL command
      ResultSet resultSet = statement.executeQuery(sqlCommand);

      // Find the number of columns in the result set
      int columnCount = resultSet.getMetaData().getColumnCount();
      String row = "";

      // Display column names
      for (int i = 1; i <= columnCount; i++) {
        row += resultSet.getMetaData().getColumnName(i) + "\t";
      }

      taSQLResult.appendText(row + '\n');

      while (resultSet.next()) {
        // Reset row to empty
        row = "";

        for (int i = 1; i <= columnCount; i++) {
          // A non-String column is converted to a string
          row += resultSet.getString(i) + "\t";
        }

        taSQLResult.appendText(row + '\n');
      }
    }
    catch (SQLException ex) {
      taSQLResult.setText(ex.toString());
    }
  }

```

```
200    /** Execute SQL DDL, and modification commands */
201    private void processSQLNonSelect(String sqlCommand) {
202      try {
203        // Get a new statement for the current connection
204        statement = connection.createStatement();
205
206        // Execute a non-SELECT SQL command
207        statement.executeUpdate(sqlCommand);
208
209        taSQLResult.setText("SQL command executed");
210      }
211      catch (SQLException ex) {
212        taSQLResult.setText(ex.toString());
213      }
214    }
215  }
```

The user selects or enters the JDBC driver, database URL, username, and password, and clicks the *Connect to Database* button to connect to the specified database using the `connectToDB()` method (lines 122–140).

When the user clicks the *Execute SQL Command* button, the `executeSQL()` method is invoked (lines 143–161) to get the SQL commands from the text area (`jtaSQLCommand`) and extract each command separated by a semicolon (`;`). It then determines whether the command is a SELECT query or a DDL or data modification statement (lines 153–158). If the command is a SELECT query, the `processSQLSelect` method is invoked (lines 164–198). This method uses the `executeQuery` method (line 170) to obtain the query result. The result is displayed in the text area `jtaSQLResult` (line 181). If the command is a non-SELECT query, the `processSQLNonSelect()` method is invoked (lines 201–214). This method uses the `executeUpdate` method (line 207) to execute the SQL command.

The `getMetaData` method (lines 173, 178) in the `ResultSet` interface is used to obtain an instance of `ResultSetMetaData`. The `getColumnCount` method (line 173) returns the number of columns in the result set, and the `getColumnName(i)` method (line 178) returns the column name for the *i*th column.

35.3 Batch Processing

Key Point

You can send a batch of SQL statements to the database for execution at once to improve efficiency.

In all the preceding examples, SQL commands are submitted to the database for execution one at a time. This is inefficient for processing a large number of updates. For example, suppose you wanted to insert a thousand rows into a table. Submitting one INSERT command at a time would take nearly a thousand times longer than submitting all the INSERT commands in a batch at once. To improve performance, JDBC introduced the batch update for processing nonselect SQL commands. A batch update consists of a sequence of nonselect SQL commands. These commands are collected in a batch and submitted to the database all together.

To use the batch update, you add nonselect commands to a batch using the `addBatch` method in the `Statement` interface. After all the SQL commands are added to the batch, use the `executeBatch` method to submit the batch to the database for execution.

For example, the following code adds a create table command, adds two insert statements in a batch, and executes the batch:

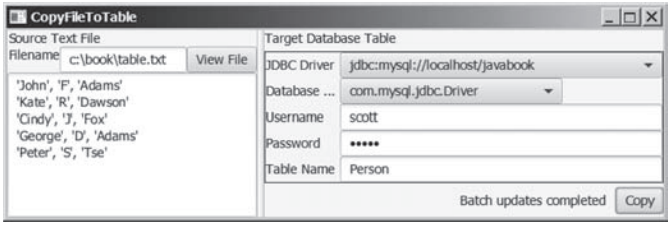
```
Statement statement = connection.createStatement();

// Add SQL commands to the batch
statement.addBatch("create table T (C1 integer, C2 varchar(15))");
statement.addBatch("insert into T values (100, 'Smith')");
statement.addBatch("insert into T values (200, 'Jones')");

// Execute the batch
int count[] = statement.executeBatch();
```

The `executeBatch()` method returns an array of counts, each of which counts the number of rows affected by the SQL command. The first count returns `0` because it is a DDL command. The other counts return `1` because only one row is affected.

Note
To find out whether a driver supports batch updates, invoke `supportsBatchUpdates()` on a `DatabaseMetaData` instance. If the driver supports batch updates, it will return `true`. The JDBC drivers for MySQL, Access, and Oracle all support batch updates.

To demonstrate batch processing, consider writing a program that gets data from a text file and copies the data from the text file to a table, as shown in Figure 35.2. The text file consists of lines that each corresponds to a row in the table. The fields in a row are separated by commas. The string values in a row are enclosed in single quotes. You can view the text file by clicking the *View File* button and copy the text to the table by clicking the *Copy* button. The table must already be defined in the database. Figure 35.2 shows the text file `table.txt` copied to table `Person`. `Person` is created using the following statement:

```
create table Person (
  firstName varchar(20),
  mi char(1),
  lastName varchar(20)
)
```



Filename: c:\book\table.txt

```
'John', 'F', 'Adams'
'Kate', 'R', 'Dawson'
'Cindy', 'J', 'Fox'
'George', 'D', 'Adams'
'Peter', 'S', 'Tse'
```

JDBC Driver: jdbc:mysql://localhost/javabook
Database ...: com.mysql.jdbc.Driver
Username: scott
Password: •••••
Table Name: Person

Batch updates completed

FIGURE 35.2 The `CopyFileToTable` utility copies text files to database tables.

Listing 35.2 gives the solution to the problem.

LISTING 35.2 CopyFileToTable.java

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.sql.*;
import java.util.Scanner;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.control.PasswordField;
import javafx.scene.control.SplitPane;
```

```java
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class CopyFileToTable extends Application {
  // Text file info
  private TextField tfFilename = new TextField();
  private TextArea taFile = new TextArea();

  // JDBC and table info
  private ComboBox<String> cboURL = new ComboBox<>();
  private ComboBox<String> cboDriver = new ComboBox<>();
  private TextField tfUsername = new TextField();
  private PasswordField pfPassword = new PasswordField();
  private TextField tfTableName = new TextField();

  private Button btViewFile = new Button("View File");
  private Button btCopy = new Button("Copy");
  private Label lblStatus = new Label();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    cboURL.getItems().addAll(FXCollections.observableArrayList(
      "jdbc:mysql://localhost/javabook",
      "jdbc:mysql://liang.armstrong.edu/javabook",
      "jdbc:odbc:exampleMDBDataSource",
      "jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl"));
    cboURL.getSelectionModel().selectFirst();

    cboDriver.getItems().addAll(FXCollections.observableArrayList(
      "com.mysql.jdbc.Driver", "sun.jdbc.odbc.dbcOdbcDriver",
      "oracle.jdbc.driver.OracleDriver"));
    cboDriver.getSelectionModel().selectFirst();

    // Create UI for connecting to the database
    GridPane gridPane = new GridPane();
    gridPane.add(new Label("JDBC Driver"), 0, 0);
    gridPane.add(new Label("Database URL"), 0, 1);
    gridPane.add(new Label("Username"), 0, 2);
    gridPane.add(new Label("Password"), 0, 3);
    gridPane.add(new Label("Table Name"), 0, 4);
    gridPane.add(cboURL, 1, 0);
    gridPane.add(cboDriver, 1, 1);
    gridPane.add(tfUsername, 1, 2);
    gridPane.add(pfPassword, 1, 3);
    gridPane.add(tfTableName, 1, 4);

    HBox hBoxConnection = new HBox(10);
    hBoxConnection.getChildren().addAll(lblStatus, btCopy);
    hBoxConnection.setAlignment(Pos.CENTER_RIGHT);

    VBox vBoxConnection = new VBox(5);
    vBoxConnection.getChildren().addAll(
      new Label("Target Database Table"),
      gridPane, hBoxConnection);

    gridPane.setStyle("-fx-border-color: black;");

    BorderPane borderPaneFileName = new BorderPane();
    borderPaneFileName.setLeft(new Label("Filename"));
    borderPaneFileName.setCenter(tfFilename);
    borderPaneFileName.setRight(btViewFile);

    BorderPane borderPaneFileContent = new BorderPane();
    borderPaneFileContent.setTop(borderPaneFileName);
    borderPaneFileContent.setCenter(taFile);

    BorderPane borderPaneFileSource = new BorderPane();
    borderPaneFileSource.setTop(new Label("Source Text File"));
    borderPaneFileSource.setCenter(borderPaneFileContent);

    SplitPane sp = new SplitPane();
    sp.getItems().addAll(borderPaneFileSource, vBoxConnection);

    // Create a scene and place it in the stage
    Scene scene = new Scene(sp, 680, 230);
    primaryStage.setTitle("CopyFileToTable"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    btViewFile.setOnAction(e -> showFile());
    btCopy.setOnAction(e -> {
      try {
        copyFile();
      }
      catch (Exception ex) {
        lblStatus.setText(ex.toString());
      }
    });
  }

  /** Display the file in the text area */
  private void showFile() {
    Scanner input = null;
    try {
      // Use a Scanner to read text from the file
      input = new Scanner(new File(tfFilename.getText().trim()));

      // Read a line and append the line to the text area
      while (input.hasNext())
        taFile.appendText(input.nextLine() + '\n');
    }
    catch (FileNotFoundException ex) {
      System.out.println("File not found: " + tfFilename.getText());
    }
    catch (IOException ex) {
      ex.printStackTrace();
    }
    finally {
      if (input != null) input.close();
    }
  }

  private void copyFile() throws Exception {
    // Load the JDBC driver
    Class.forName(cboDriver.getSelectionModel()
      .getSelectedItem().trim());
    System.out.println("Driver loaded");
```

```

      // Establish a connection
      Connection conn = DriverManager.getConnection(
        cboURL.getSelectionModel().getSelectedItem().trim(),
        tfUsername.getText().trim(),
        String.valueOf(pfPassword.getText()).trim());
      System.out.println("Database connected");

      // Read each line from the text file and insert it to the table
      insertRows(conn);
    }

    private void insertRows(Connection connection) {
      // Build the SQL INSERT statement
      String sqlInsert = "insert into " + tfTableName.getText()
        + " values (";

      // Use a Scanner to read text from the file
      Scanner input = null;

      // Get file name from the text field
      String filename = tfFilename.getText().trim();

      try {
        // Create a scanner
        input = new Scanner(new File(filename));

        // Create a statement
        Statement statement = connection.createStatement();

        System.out.println("Driver major version? " +
          connection.getMetaData().getDriverMajorVersion());

        // Determine if batchUpdatesSupported is supported
        boolean batchUpdatesSupported = false;

        try {
          if (connection.getMetaData().supportsBatchUpdates()) {
            batchUpdatesSupported = true;
            System.out.println("batch updates supported");
          }
          else {
            System.out.println("The driver " +
              "does not support batch updates");
          }
        }
        catch (UnsupportedOperationException ex) {
          System.out.println("The operation is not supported");
        }

        // Determine if the driver is capable of batch updates
        if (batchUpdatesSupported) {
          // Read a line and add the insert table command to the batch
          while (input.hasNext()) {
            statement.addBatch(sqlInsert + input.nextLine() + ")");
          }

          statement.executeBatch();

          lblStatus.setText("Batch updates completed");
        }
```

```
198        else {
199          // Read a line and execute insert table command
200          while (input.hasNext()) {
201            statement.executeUpdate(sqlInsert + input.nextLine() + ")");
202          }
203
204          lblStatus.setText("Single row update completed");
205        }
206      }
207      catch (SQLException ex) {
208        System.out.println(ex);
209      }
210      catch (FileNotFoundException ex) {
211        System.out.println("File not found: " + filename);
212      }
213      finally {
214        if (input != null) input.close();
215      }
216    }
217  }
```

The insertRows method (lines 149–216) uses the batch updates to submit SQL INSERT commands to the database for execution, if the driver supports batch updates. Lines 174–181 check whether the driver supports batch updates. If the driver does not support the operation, an UnsupportedOperationException exception will be thrown (line 183) when the supportsBatchUpdates() method is invoked.

The tables must already be created in the database. The file format and contents must match the database table specification. Otherwise, the SQL INSERT command will fail.

In Exercise 35.1, you will write a program to insert a thousand records to a database and compare the performance with and without batch updates.

Check Point

35.3.1 What is batch processing in JDBC? What are the benefits of using batch processing?

35.3.2 How do you add an SQL statement to a batch? How do you execute a batch?

35.3.3 Can you execute a SELECT statement in a batch?

35.3.4 How do you know whether a JDBC driver supports batch updates?

35.4 Scrollable and Updatable Result Set

Key Point

You can use scrollable and updatable result set to move the cursor anywhere in the result set to perform insertion, deletion, and update.

The result sets used in the preceding examples are read sequentially. A result set maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The next() method moves the cursor forward to the next row. This is known as *sequential forward reading*.

A more powerful way of accessing database is to use a scrollable and updatable result, which enables you to scroll the rows both forward and backward and move the cursor to a desired location using the first, last, next, previous, absolute, or relative method. Additionally, you can insert, delete, or update a row in the result set and have the changes automatically reflected in the database.

To obtain a scrollable or updatable result set, you must first create a statement with an appropriate type and concurrency mode. For a static statement, use

```
Statement statement = connection.createStatement
  (int resultSetType, int resultSetConcurrency);
```

For a prepared statement, use

```
PreparedStatement statement = connection.prepareStatement
  (String sql, int resultSetType, int resultSetConcurrency);
```

The possible values of `resultSetType` are the constants defined in the `ResultSet`:

- `TYPE_FORWARD_ONLY`: The result set is accessed forward sequentially.
- `TYPE_SCROLL_INSENSITIVE`: The result set is scrollable, but not sensitive to changes in the database.
- `TYPE_SCROLL_SENSITIVE`: The result set is scrollable and sensitive to changes made by others. Use this type if you want the result set to be scrollable and updatable.

The possible values of `resultSetConcurrency` are the constants defined in the `ResultSet`:

- `CONCUR_READ_ONLY`: The result set cannot be used to update the database.
- `CONCUR_UPDATABLE`: The result set can be used to update the database.

For example, if you want the result set to be scrollable and updatable, you can create a statement, as follows:

```
Statement statement = connection.createStatement
  (ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE)
```

You use the `executeQuery` method in a `Statement` object to execute an SQL query that returns a result set as follows:

```
ResultSet resultSet = statement.executeQuery(query);
```

You can now use the methods `first()`, `next()`, `previous()`, and `last()` to move the cursor to the first row, next row, previous row, and last row. The `absolute(int row)` method moves the cursor to the specified row; and the `get`*`Xxx`*`(int columnIndex)` or `get`*`Xxx`*`(String columnName)` method is used to retrieve the value of a specified field at the current row. The methods `insertRow()`, `deleteRow()`, and `updateRow()` can also be used to insert, delete, and update the current row. Before applying `insertRow` or `updateRow`, you need to use the method `updateXxx(int columnIndex,` *`Xxx`* `value)` or `update(String columnName,` *`Xxx`* `value)` to write a new value to the field at the current row. The `cancelRowUpdates()` method cancels the updates made to a row. The `close()` method closes the result set and releases its resource. The `wasNull()` method returns true if the last column read had a value of SQL NULL.

Listing 35.3 gives an example that demonstrates how to create a scrollable and updatable result set. The program creates a result set for the `StateCapital` table. The `StateCapital` table is defined as follows:

```
create table StateCapital (
  state varchar(40),
  capital varchar(40)
);
```

LISTING 35.3 ScrollUpdateResultSet.java

```
import java.sql.*;

public class ScrollUpdateResultSet {
  public static void main(String[] args)
      throws SQLException, ClassNotFoundException {
```

```
    // Load the JDBC driver
    Class.forName("oracle.jdbc.driver.OracleDriver");
    System.out.println("Driver loaded");

    // Connect to a database
    Connection connection = DriverManager.getConnection
      ("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl",
       "scott", "tiger");
    connection.setAutoCommit(true);
    System.out.println("Database connected");

    // Get a new statement for the current connection
    Statement statement = connection.createStatement(
      ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);

    // Get ResultSet
    ResultSet resultSet = statement.executeQuery
      ("select state, capital from StateCapital");

    System.out.println("Before update ");
    displayResultSet(resultSet);

    // Update the second row
    resultSet.absolute(2); // Move cursor to the second row
    resultSet.updateString("state", "New S"); // Update the column
    resultSet.updateString("capital", "New C"); // Update the column
    resultSet.updateRow(); // Update the row in the data source

    // Insert after the last row
    resultSet.last();
    resultSet.moveToInsertRow(); // Move cursor to the insert row
    resultSet.updateString("state", "Florida");
    resultSet.updateString("capital", "Tallahassee");
    resultSet.insertRow(); // Insert the row
    resultSet.moveToCurrentRow(); // Move the cursor to the current row

    // Delete fourth row
    resultSet.absolute(4); // Move cursor to the 5th row
    resultSet.deleteRow(); // Delete the second row

    System.out.println("After update ");
    resultSet = statement.executeQuery
      ("select state, capital from StateCapital");
    displayResultSet(resultSet);

    // Close the connection
    resultSet.close();
  }

  private static void displayResultSet(ResultSet resultSet)
      throws SQLException {
    ResultSetMetaData rsMetaData = resultSet.getMetaData();
    resultSet.beforeFirst();
    while (resultSet.next()) {
      for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
        System.out.printf("%-12s\t", resultSet.getObject(i));
      System.out.println();
    }
  }
}
```

```
Driver loaded
Database connected

Before update
Indiana            Indianapolis
Illinois           Springfield
California         Sacramento
Georgia            Atlanta
Texas              Austin

After update
Indiana            Indianapolis
New S              New C
California         Sacramento
Texas              Austin
Florida            Tallahassee
```

The code in lines 18–19 creates a `Statement` for producing scrollable and updatable result sets.

The program moves the cursor to the second row in the result set (line 29), updates two columns in this row (lines 30–31), and invokes the `updateRow()` method to update the row in the underlying database (line 32).

An updatable `ResultSet` object has a special row associated with it that serves as a staging area for building a row to be inserted. This special row is called the *insert row*. To insert a row, first invoke the `moveToInsertRow()` method to move the cursor to the insert row (line 36), then update the columns using the `updateXxx` method (lines 37–38), and finally insert the row using the `insertRow()` method (line 39). Invoking `moveToCurrentRow()` moves the cursor to the current inserted row (lines 40).

The program moves to the fourth row and invokes the `deleteRow()` method to delete the row from the database (lines 43–44).

Note

Not all current drivers support scrollable and updatable result sets. The example is tested using Oracle ojdbc6 driver. You can use `supportsResultSetType(int type)` and `supportsResultSetConcurrency(int type, int concurrency)` in the `DatabaseMetaData` interface to find out which result type and currency modes are supported by the JDBC driver. But even if a driver supports the scrollable and updatable result set, a result set for a complex query might not be able to perform an update. For example, the result set for a query that involves several tables is likely not to support update operations.

Note

The program may not work due to an issue in the Oracle JDBC driver if lines 22–23 are replaced by

```
ResultSet resultSet = statement.executeQuery
  ("select * from StateCapital");
```

Check Point

35.4.1 What is a scrollable result set? What is an updatable result set?

35.4.2 How do you create a scrollable and updatable `ResultSet`?

35.4.3 How do you know whether a JDBC driver supports a scrollable and updatable `ResultSet`?

35.5 RowSet, JdbcRowSet, and CachedRowSet

The RowSet interface can be used to simplify database programming.

Key Point

The RowSet interface extends java.sql.ResultSet with additional capabilities that allow a RowSet instance to be configured to connect to a JDBC url, username, and password, set an SQL command, execute the command, and retrieve the execution result. In essence, it combines Connection, Statement, and ResultSet into one interface.

Note
Not all JDBC drivers support RowSet. Currently, the JDBC-ODBC driver does not support all features of RowSet.

35.5.1 RowSet Basics

There are two types of RowSet objects: connected and disconnected. A *connected* RowSet object makes a connection with a data source and maintains that connection throughout its life cycle. A disconnected RowSet object makes a connection with a data source, executes a query to get data from the data source, and then closes the connection. A *disconnected* rowset may make changes to its data while it is disconnected and then send the changes back to the original source of the data, but it must reestablish a connection to do so.

There are several versions of RowSet. Two frequently used are JdbcRowSet and CachedRowSet. Both are subinterfaces of RowSet. JdbcRowSet is connected, while CachedRowSet is disconnected. Also, JdbcRowSet is neither serializable nor cloneable, while CachedRowSet is both. The database vendors are free to provide concrete implementations for these interfaces. Oracle has provided the reference implementation JdbcRowSetImpl for JdbcRowSet and CachedRowSetImpl for CachedRowSet. Figure 35.3 shows the relationship of these components.

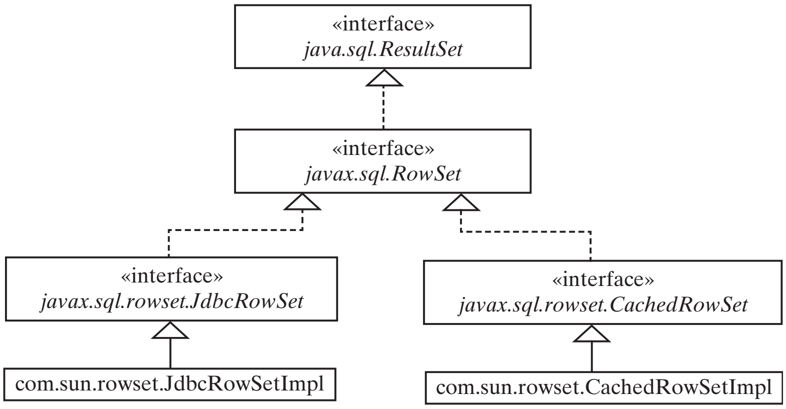

«interface»
java.sql.ResultSet
«interface»
javax.sql.RowSet
«interface»
javax.sql.rowset.JdbcRowSet
«interface»
javax.sql.rowset.CachedRowSet
com.sun.rowset.JdbcRowSetImpl
com.sun.rowset.CachedRowSetImpl

Figure 35.3 The JdbcRowSetImpl and CachedRowSetImpl are concrete implementations of RowSet.

The RowSet interface contains the JavaBeans properties with getter and setter methods. You can use the setter methods to set a new url, username, password, and command for an SQL statement. Using a RowSet, Listing 34.1 can be simplified, as shown in Listing 35.4.

Listing 35.4 SimpleRowSet.java

```
import java.sql.SQLException;
import javax.sql.RowSet;
import com.sun.rowset.*;

```

```
 5  public class SimpleRowSet {
 6    public static void main(String[] args)
 7        throws SQLException, ClassNotFoundException {
 8      // Load the JDBC driver
 9      Class.forName("com.mysql.jdbc.Driver");
10      System.out.println("Driver loaded");
11
12      // Create a row set
13      RowSet rowSet = new JdbcRowSetImpl();
14
15      // Set RowSet properties
16      rowSet.setUrl("jdbc:mysql://localhost/javabook");
17      rowSet.setUsername("scott");
18      rowSet.setPassword("tiger");
19      rowSet.setCommand("select firstName, mi, lastName " +
20        "from Student where lastName = 'Smith'");
21      rowSet.execute();
22
23      // Iterate through the result and print the student names
24      while (rowSet.next())
25        System.out.println(rowSet.getString(1) + "\t" +
26          rowSet.getString(2) + "\t" + rowSet.getString(3));
27
28      // Close the connection
29      rowSet.close();
30    }
31  }
```

Line 13 creates a RowSet object using JdbcRowSetImpl. The program uses the RowSet's set method to set a URL, username, and password (lines 16–18) and a command for a query statement (line 19). Line 24 executes the command in the RowSet. The methods next() and getString(int) for processing the query result (lines 25–26) are inherited from ResultSet.

If you replace JdbcRowSet with CachedRowSet in line 13, the program will work just fine. Note, the JDBC-ODBC driver supports JdbcRowSetImpl, but not CachedRowSetImpl.

Tip

Since RowSet is a subinterface of ResultSet, all the methods in ResultSet can be used in RowSet. For example, you can obtain ResultSetMetaData from a RowSet using the getMetaData() method.

35.5.2 RowSet for PreparedStatement

The discussion in §34.5, "PreparedStatement," introduced processing parameterized SQL statements using the PreparedStatement interface. RowSet has the capability to support parameterized SQL statements. The set methods for setting parameter values in PreparedStatement are implemented in RowSet. You can use these methods to set parameter values for a parameterized SQL command. Listing 35.5 demonstrates how to use a parameterized statement in RowSet. Line 19 sets an SQL query statement with two parameters for lastName and mi in a RowSet. Since these two parameters are strings, the setString method is used to set actual values in lines 21–22.

LISTING 35.5 RowSetPreparedStatement.java

```
1  import java.sql.*;
2  import javax.sql.RowSet;
3  import com.sun.rowset.*;
4
```

```
public class RowSetPreparedStatement {
  public static void main(String[] args)
      throws SQLException, ClassNotFoundException {
    // Load the JDBC driver
    Class.forName("com.mysql.jdbc.Driver");
    System.out.println("Driver loaded");

    // Create a row set
    RowSet rowSet = new JdbcRowSetImpl();

    // Set RowSet properties
    rowSet.setUrl("jdbc:mysql://localhost/javabook");
    rowSet.setUsername("scott");
    rowSet.setPassword("tiger");
    rowSet.setCommand("select * from Student where lastName = ? " +
      "and mi = ?");
    rowSet.setString(1, "Smith");
    rowSet.setString(2, "R");
    rowSet.execute();

    ResultSetMetaData rsMetaData = rowSet.getMetaData();
    for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
      System.out.printf("%-12s\t", rsMetaData.getColumnName(i));
    System.out.println();

    // Iterate through the result and print the student names
    while (rowSet.next()) {
      for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
        System.out.printf("%-12s\t", rowSet.getObject(i));
      System.out.println();
    }

    // Close the connection
    rowSet.close();
  }
}
```

35.5.3 Scrolling and Updating RowSet

By default, a ResultSet object is neither scrollable nor updatable. However, a RowSet object is both. It is easier to scroll and update a database through a RowSet than through a ResultSet. Listing 35.6 rewrites Listing 35.3 using a RowSet. You can use methods such as absolute(int) to move the cursor and methods such as delete(), updateRow(), and insertRow() to update the database.

Listing 35.6 ScrollUpdateRowSet.java

```
import java.sql.*;
import javax.sql.RowSet;
import com.sun.rowset.JdbcRowSetImpl;

public class ScrollUpdateRowSet {
  public static void main(String[] args)
      throws SQLException, ClassNotFoundException {
    // Load the JDBC driver
    Class.forName("com.mysql.jdbc.Driver");
    System.out.println("Driver loaded");

12     // Create a row set
13     RowSet rowSet = new JdbcRowSetImpl();
14
15     // Set RowSet properties
16     rowSet.setUrl("jdbc:mysql://localhost/javabook");
17     rowSet.setUsername("scott");
18     rowSet.setPassword("tiger");
19     rowSet.setCommand("select state, capital from StateCapital");
20     rowSet.execute();
21
22     System.out.println("Before update ");
23     displayRowSet(rowSet);
24
25     // Update the second row
26     rowSet.absolute(2); // Move cursor to the 2nd row
27     rowSet.updateString("state", "New S"); // Update the column
28     rowSet.updateString("capital", "New C"); // Update the column
29     rowSet.updateRow(); // Update the row in the data source
30
31     // Insert after the second row
32     rowSet.last();
33     rowSet.moveToInsertRow(); // Move cursor to the insert row
34     rowSet.updateString("state", "Florida");
35     rowSet.updateString("capital", "Tallahassee");
36     rowSet.insertRow(); // Insert the row
37     rowSet.moveToCurrentRow(); // Move the cursor to the current row
38
39     // Delete fourth row
40     rowSet.absolute(4); // Move cursor to the fifth row
41     rowSet.deleteRow(); // Delete the second row
42
43     System.out.println("After update ");
44     displayRowSet(rowSet);
45
46     // Close the connection
47     rowSet.close();
48   }
49
50   private static void displayRowSet(RowSet rowSet)
51       throws SQLException {
52     ResultSetMetaData rsMetaData = rowSet.getMetaData();
53     rowSet.beforeFirst();
54     while (rowSet.next()) {
55       for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
56         System.out.printf("%-12s\t", rowSet.getObject(i));
57       System.out.println();
58     }
59   }
60 }
```

If you replace JdbcRowSet with CachedRowSet in line 13, the database is not changed. To make the changes on the CachedRowSet effective in the database, you must invoke the acceptChanges() method after you make all the changes, as follows:

```
// Write changes back to the database
((com.sun.rowset.CachedRowSetImpl)rowSet).acceptChanges();
```

This method automatically reconnects to the database and writes all the changes back to the database.

35.5.4 RowSetEvent

A RowSet object fires a RowSetEvent whenever the object's cursor has moved, a row has changed, or the entire row set has changed. This event can be used to synchronize a RowSet with the components that rely on the RowSet. For example, a visual component that displays the contents of a RowSet should be synchronized with the RowSet. The RowSetEvent can be used to achieve synchronization. The handlers in RowSetListener are cursorMoved(RowSetEvent), rowChanged(RowSetEvent), and cursorSetChanged(RowSetEvent).

Listing 35.7 gives an example that demonstrates RowSetEvent. A listener for RowSetEvent is registered in lines 14–26. When rowSet.execute() (line 33) is executed, the entire row set is changed, so the listener's rowSetChanged handler is invoked. When rowSet.last() (line 35) is executed, the cursor is moved, so the listener's cursorMoved handler is invoked. When rowSet.updateRow() (line 37) is executed, the row is updated, so the listener's rowChanged handler is invoked.

LISTING 35.7 TestRowSetEvent.java

```
 1  import java.sql.*;
 2  import javax.sql.*;
 3  import com.sun.rowset.*;
 4
 5  public class TestRowSetEvent {
 6    public static void main(String[] args)
 7        throws SQLException, ClassNotFoundException {
 8      // Load the JDBC driver
 9      Class.forName("com.mysql.jdbc.Driver");
10      System.out.println("Driver loaded");
11
12      // Create a row set
13      RowSet rowSet = new JdbcRowSetImpl();
14      rowSet.addRowSetListener(new RowSetListener() {
15        public void cursorMoved(RowSetEvent e) {
16          System.out.println("Cursor moved");
17        }
18
19        public void rowChanged(RowSetEvent e) {
20          System.out.println("Row changed");
21        }
22
23        public void rowSetChanged(RowSetEvent e) {
24          System.out.println("row set changed");
25        }
26      });
27
28      // Set RowSet properties
29      rowSet.setUrl("jdbc:mysql://localhost/javabook");
30      rowSet.setUsername("scott");
31      rowSet.setPassword("tiger");
32      rowSet.setCommand("select * from Student");
33      rowSet.execute();
34
35      rowSet.last(); // Cursor moved
36      rowSet.updateString("lastName", "Yao"); // Update column
37      rowSet.updateRow(); // Row updated
38
39      // Close the connection
40      rowSet.close();
41    }
42  }
```

Check Point

35.5.1 What are the advantages of RowSet?

35.5.2 What are JdbcRowSet and CachedRowSet? What are the differences between them?

35.5.3 How do you create a JdbcRowSet and a CachedRowSet?

35.5.4 Can you scroll and update a RowSet? What method must be invoked to write the changes in a CachedRowSet to the database?

35.5.5 Describe the handlers in RowSetListener.

35.6 Storing and Retrieving Images in JDBC

Key Point

You can store and retrieve images using JDBC.

A database can store not only numbers and strings, but also images. SQL3 introduced a new data type called BLOB (*B*inary *L*arge *OB*ject) for storing binary data, which can be used to store images. Another new SQL3 type is CLOB (*C*haracter *L*arge *OB*ject) for storing a large text in the character format. JDBC introduced the interfaces java.sql.Blob and java.sql.Clob to support mapping for these new SQL types. You can use getBlob, setBinaryStream, getClob, setBlob, and setClob, to access SQL BLOB and CLOB values in the interfaces ResultSet and PreparedStatement.

To store an image into a cell in a table, the corresponding column for the cell must be of the BLOB type. For example, the following SQL statement creates a table whose type for the flag column is BLOB:

```
create table Country(name varchar(30), flag blob,
  description varchar(255));
```

In the preceding statement, the description column is limited to 255 characters, which is the upper limit for MySQL. For Oracle, the upper limit is 32,672 bytes. For a large character field, you can use the CLOB type for Oracle, which can store up to two GB of characters. MySQL does not support CLOB. However, you can use BLOB to store a long string and convert binary data into characters.

Note

MS Access database does not support the BLOB and CLOB types.

To insert a record with images to a table, define a prepared statement like this one:

```
  PreparedStatement pstmt = connection.prepareStatement(
   "insert into Country values(?, ?, ?)");
```

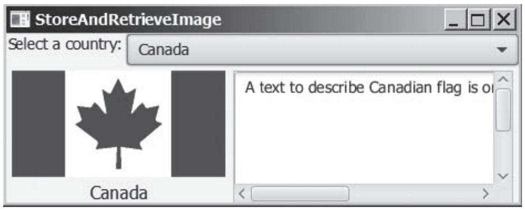
Images are usually stored in files. You may first get an instance of InputStream for an image file then use the setBinaryStream method to associate the input stream with a cell in the table, as follows:

```
// Store image to the table cell
File file = new File(imageFilename);
InputStream inputImage = new FileInputStream(file);
pstmt.setBinaryStream(2, inputImage, (int)(file.length()));
```

To retrieve an image from a table, use the getBlob method, as shown below:

```
// Store image to the table cell
Blob blob = rs.getBlob(1);
ImageIcon imageIcon = new ImageIcon(
  blob.getBytes(1, (int)blob.length()));
```

Listing 35.8 gives a program that demonstrates how to store and retrieve images in JDBC. The program first creates the Country table and stores data to it. Then the program retrieves the country names from the table and adds them to a combo box. When the user selects a name from the combo box, the country's flag and description are displayed, as shown in Figure 35.4.



A text to describe Canadian flag is o

FIGURE 35.4 The program enables you to retrieve data, including images, from a table and displays them.

LISTING 35.8 StoreAndRetrieveImage.java

```
import java.sql.*;
import java.io.*;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class StoreAndRetrieveImage extends Application {
  // Connection to the database
  private Connection connection;

  // Statement for static SQL statements
  private Statement stmt;

  // Prepared statement
  private PreparedStatement pstmt = null;
  private DescriptionPane descriptionPane
    = new DescriptionPane();

  private ComboBox<String> cboCountry = new ComboBox<>();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    try {
      connectDB(); // Connect to DB
      storeDataToTable(); //Store data to the table (including image)
      fillDataInComboBox(); // Fill in combo box
      retrieveFlagInfo(cboCountry.getSelectionModel().getSelectedItem());
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }

    BorderPane paneForComboBox = new BorderPane();
    paneForComboBox.setLeft(new Label("Select a country: "));
    paneForComboBox.setCenter(cboCountry);
    cboCountry.setPrefWidth(400);
    BorderPane pane = new BorderPane();
    pane.setTop(paneForComboBox);
    pane.setCenter(descriptionPane);

    Scene scene = new Scene(pane, 350, 150);
    primaryStage.setTitle("StoreAndRetrieveImage");
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    cboCountry.setOnAction(e ->
      retrieveFlagInfo(cboCountry.getValue()));
  }

  private void connectDB() throws Exception {
    // Load the driver
    Class.forName("com.mysql.jdbc.Driver");
    System.out.println("Driver loaded");

    // Establish connection
    connection = DriverManager.getConnection
      ("jdbc:mysql://localhost/javabook", "scott", "tiger");
    System.out.println("Database connected");

    // Create a statement for static SQL
    stmt = connection.createStatement();

    // Create a prepared statement to retrieve flag and description
    pstmt = connection.prepareStatement("select flag, description " +
      "from Country where name = ?");
  }

  private void storeDataToTable() {
    String[] countries = {"Canada", "UK", "USA", "Germany",
      "Indian", "China"};

   String[] imageFilenames = {"image/ca.gif", "image/uk.gif",
      "image/us.gif", "image/germany.gif", "image/india.gif",
      "image/china.gif"};

   String[] descriptions = {"A text to describe Canadian " +
      "flag is omitted", "British flag ...", "American flag ...",
      "German flag ...", "Indian flag ...", "Chinese flag ..."};

    try {
      // Create a prepared statement to insert records
      PreparedStatement pstmt = connection.prepareStatement(
        "insert into Country values(?, ?, ?)");

      // Store all predefined records
      for (int i = 0; i < countries.length; i++) {
        pstmt.setString(1, countries[i]);

        // Store image to the table cell
        java.net.URL url =
          this.getClass().getResource(imageFilenames[i]);
        InputStream inputImage = url.openStream();
        pstmt.setBinaryStream(2, inputImage,
          (int)(inputImage.available()));

        pstmt.setString(3, descriptions[i]);
        pstmt.executeUpdate();
```

```
103         }
104
105         System.out.println("Table Country populated");
106       }
107       catch (Exception ex) {
108         ex.printStackTrace();
109       }
110     }
111
112     private void fillDataInComboBox()  throws Exception {
113       ResultSet rs = stmt.executeQuery("select name from Country");
114       while (rs.next()) {
115         cboCountry.getItems().add(rs.getString(1));
116       }
117       cboCountry.getSelectionModel().selectFirst();
118     }
119
120     private void retrieveFlagInfo(String name) {
121       try {
122         pstmt.setString(1, name);
123         ResultSet rs = pstmt.executeQuery();
124         if (rs.next()) {
125           Blob blob = rs.getBlob(1);
126           ByteArrayInputStream in = new ByteArrayInputStream
127             (blob.getBytes(1, (int)blob.length()));
128           Image image = new Image(in);
129           ImageView imageView = new ImageView(image);
130           descriptionPane.setImageView(imageView);
131           descriptionPane.setTitle(name);
132           String description = rs.getString(2);
133           descriptionPane.setDescription(description);
134         }
135       }
136       catch (Exception ex) {
137         System.err.println(ex);
138       }
139     }
140   }
```

`DescriptionPane` (line 21) is a component for displaying a country (name, flag, and description). This component was presented in Listing 16.6, DescriptionPane.java.

The `storeDataToTable` method (lines 73–110) populates the table with data. The `fillDataInComboBox` method (lines 112–118) retrieves the country names and adds them to the combo box. The `retrieveFlagInfo(name)` method (lines 120–139) retrieves the flag and description for the specified country name.

Check Point

35.6.1 How do you store images into a database?

35.6.2 How do you retrieve images from a database?

35.6.3 Does Oracle support the SQL3 BLOB type and CLOB type? What about MySQL and Access?

Key Terms

BLOB type	35-20
CLOB type	35-20
batch mode	35-2
cached row set	35-15
row set	35-15
scrollable result set	35-2
updatable result set	35-11

Chapter Summary

1. This chapter developed a universal SQL client that can be used to access any local or remote relational database.

2. You can use the `addBatch(SQLString)` method to add SQL statements to a statement for batch processing.

3. You can create a statement to specify that the result set be scrollable and updatable. By default, the result set is neither of these.

4. The `RowSet` can be used to simplify Java database programming. A `RowSet` object is scrollable and updatable. A `RowSet` can fire a `RowSetEvent`.

5. You can store and retrieve image data in JDBC using the SQL BLOB type.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

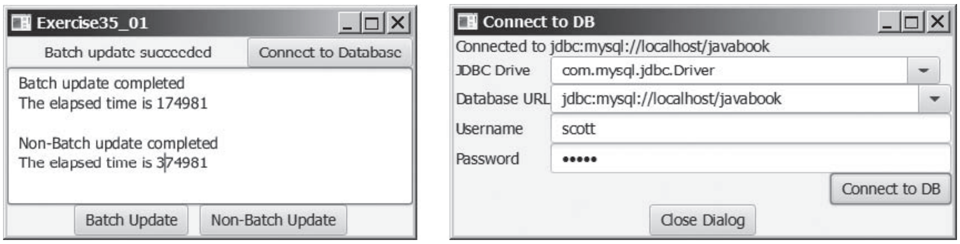
MyProgrammingLab™

Programming Exercises

***35.1** (*Batch update*) Write a program that inserts a thousand records to a database, and compare the performance with and without batch updates, as shown in Figure 35.5a. Suppose the table is defined as follows:

```
create table Temp(num1 double, num2 double, num3 double)
```

Use the `Math.random()` method to generate random numbers for each record. Create a dialog box that contains `DBConnectionPanel`, discussed in Exercise 34.3. Use this dialog box to connect to the database. When you click the *Connect to Database* button in Figure 35.5a, the dialog box in Figure 35.5b is displayed.



Batch update succeeded

```
Batch update completed
The elapsed time is 174981

Non-Batch update completed
The elapsed time is 374981
```

Connected to jdbc:mysql://localhost/javabook

JDBC Drive: com.mysql.jdbc.Driver

Database URL: jdbc:mysql://localhost/javabook

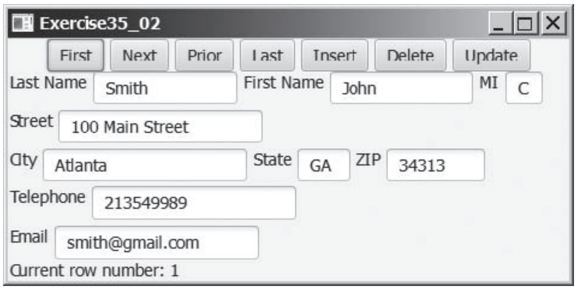
Username: scott

Password: •••••

(a) (b)

Figure 35.5 The program demonstrates the performance improvements that result from using batch updates.

****35.2** (*Scrollable result set*) Write a program that uses the buttons *First*, *Next*, *Prior*, *Last*, *Insert*, *Delete*, and *Update*, and modify a single record in the `Address` table, as shown in Figure 35.6.



Last Name Smith First Name John MI C
Street 100 Main Street
City Atlanta State GA ZIP 34313
Telephone 213549989
Email smith@gmail.com
Current row number: 1

FIGURE 35.6 You can use the buttons to display and modify a single record in the `Address` table.

The `Address` table is defined as follows:

```
create table Address (
  firstname varchar(25),
  mi char(1),
  lastname varchar(25),
  street varchar(40),
  city varchar(20),
  state varchar(2),
  zip varchar(5),
  telephone varchar(10),
  email varchar(30),
  primary key (firstname, mi, lastname)
);
```

***35.3** (*Display table contents*) Write a program that displays the content for a given table. As shown in Figure 35.7a, you enter a table and click the *Show Contents* button to display the table contents in a table view.

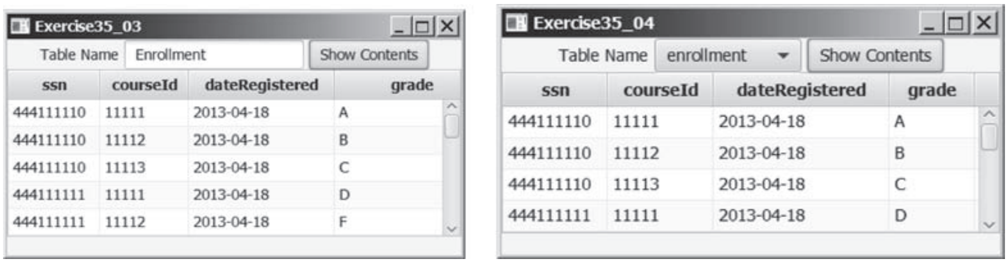


Table Name Enrollment

ssn	courseId	dateRegistered	grade
444111110	11111	2013-04-18	A
444111110	11112	2013-04-18	B
444111110	11113	2013-04-18	C
444111111	11111	2013-04-18	D
444111111	11112	2013-04-18	F

Table Name enrollment

ssn	courseId	dateRegistered	grade
444111110	11111	2013-04-18	A
444111110	11112	2013-04-18	B
444111110	11113	2013-04-18	C
444111111	11111	2013-04-18	D

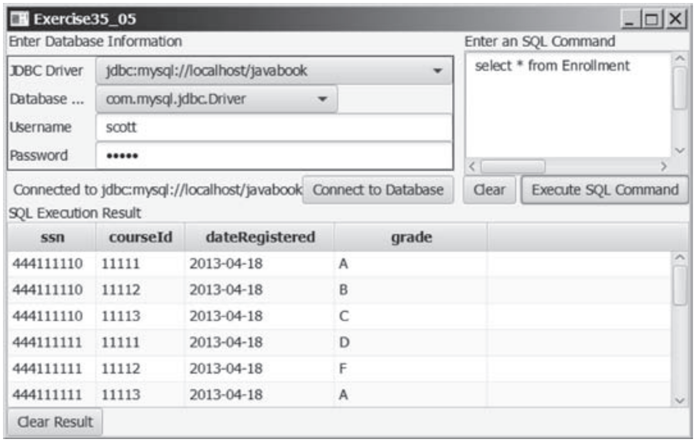
(a)

(b)

FIGURE 35.7 (a) Enter a table name to display the table contents. (b) Select a table name from the combo box to display its contents.

***35.4** (*Find tables and showing their contents*) Write a program that fills in table names in a combo box, as shown in Figure 35.7b. You can select a table from the combo box to display its contents in a table view.

****35.5** (*Revise SQLClient.java*) Rewrite Listing 35.1, SQLClient.java, to display the query result in a `TableView`, as shown in Figure 35.8.



Enter Database Information

JDBC Driver jdbc:mysql://localhost/javabook

Database ... com.mysql.jdbc.Driver

Username scott

Password •••••

Connected to jdbc:mysql://localhost/javabook

Enter an SQL Command

select * from Enrollment

SQL Execution Result

ssn	courseId	dateRegistered	grade
444111110	11111	2013-04-18	A
444111110	11112	2013-04-18	B
444111110	11113	2013-04-18	C
444111111	11111	2013-04-18	D
444111111	11112	2013-04-18	F
444111111	11113	2013-04-18	A

FIGURE 35.8 The query result is displayed in a `TableView`.

***35.5** (*Populate Salary table)* Rewrite Programming Exercise 34.8 using a batch mode to improve performance.

CHAPTER

36

Internationalization

Objectives

- To describe Java’s internationalization features (§36.1).
- To construct a locale with language, country, and variant (§36.2).
- To display date and time based on locale (§36.3).
- To display numbers, currencies, and percentages based on locale (§36.4).
- To develop applications for international audiences using resource bundles (§36.5).
- To specify encoding schemes for text I/O (§36.6).

36.1 Introduction

This chapter introduces writing Java code for international audience.

Key Point

Many websites maintain several versions of webpages so that readers can choose one written in a language they understand. Because there are so many languages in the world, it would be highly problematic to create and maintain enough different versions to meet the needs of all clients everywhere. Java comes to the rescue. Java is the first language designed from the ground up to support internationalization. In consequence, it allows your programs to be customized for any number of countries or languages without requiring cumbersome changes in the code.

Here are the major Java features that support internationalization:

- Java characters use *Unicode*, a 16-bit encoding scheme established by the Unicode Consortium to support the interchange, processing, and display of written texts in the world's diverse languages. The use of Unicode encoding makes it easy to write Java programs that can manipulate strings in any international language. (To see all the Unicode characters, visit mindprod.com/jgloss/reuters.html.)
- Java provides the `Locale` class to encapsulate information about a specific locale. A `Locale` object determines how locale-sensitive information, such as date, time, and number, is displayed, and how locale-sensitive operations, such as sorting strings, are performed. The classes for formatting date, time, and numbers, and for sorting strings are grouped in the `java.text` package.
- Java uses the `ResourceBundle` class to separate locale-specific information, such as status messages and GUI component labels, from the program. The information is stored outside the source code and can be accessed and loaded dynamically at runtime from a `ResourceBundle`, rather than hard-coded into the program.

In this chapter, you will learn how to format dates, numbers, currencies, and percentages for different regions, countries, and languages. You will also learn how to use resource bundles to define which images and strings are used by a component, depending on the user's locale and preferences.

36.2 The Locale Class

The `Locale` class defines a locale: language and nation.

Key Point

A `Locale` object represents a geographical, political, or cultural region in which a specific language or custom is used. For example, Americans speak English, and the Chinese speak Chinese. The conventions for formatting dates, numbers, currencies, and percentages may differ from one country to another. The Chinese, for instance, use year/month/day to represent the date, while Americans use month/day/year. It is important to realize that locale is not defined only by country. For example, Canadians speak either Canadian English or Canadian French, depending on which region of Canada they reside in.

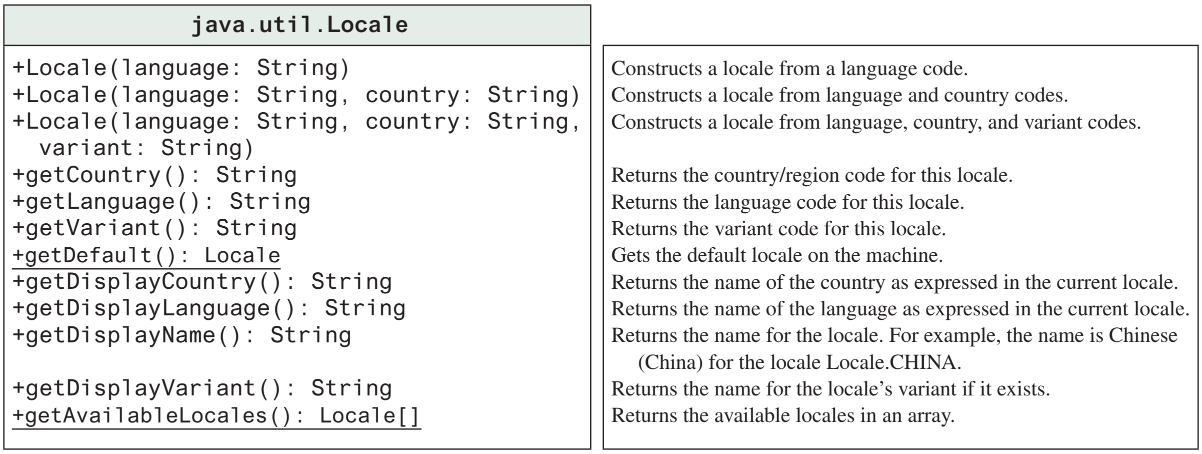
To create a `Locale` object, use one of the three constructors with a specified language and optional country and variant, as shown in Figure 36.1.

The `language` should be a valid language code—that is, one of the lowercase two-letter codes defined by ISO-639. For example, `zh` stands for Chinese, `da` for Danish, `en` for English, `de` for German, and `ko` for Korean. Table 36.1 lists the language codes.

The country should be a valid ISO country code—that is, one of the uppercase, two-letter codes defined by ISO-3166. For example, `CA` stands for Canada, `CN` for China, `DK` for Denmark, `DE` for Germany, and `US` for the United States. Table 36.2 lists the country codes.

The argument variant is rarely used and is needed only for exceptional or system-dependent situations to designate information specific to a browser or vendor. For example, the Norwegian language has two sets of spelling rules, a traditional one called *bokmål* and a new one called *nynorsk*. The locale for traditional spelling would be created as follows:

```
new Locale("no", "NO", "B");
```



java.util.Locale	
+Locale(language: String)	Constructs a locale from a language code.
+Locale(language: String, country: String)	Constructs a locale from language and country codes.
+Locale(language: String, country: String, variant: String)	Constructs a locale from language, country, and variant codes.
+getCountry(): String	Returns the country/region code for this locale.
+getLanguage(): String	Returns the language code for this locale.
+getVariant(): String	Returns the variant code for this locale.
+getDefault(): Locale	Gets the default locale on the machine.
+getDisplayCountry(): String	Returns the name of the country as expressed in the current locale.
+getDisplayLanguage(): String	Returns the name of the language as expressed in the current locale.
+getDisplayName(): String	Returns the name for the locale. For example, the name is Chinese (China) for the locale Locale.CHINA.
+getDisplayVariant(): String	Returns the name for the locale's variant if it exists.
+getAvailableLocales(): Locale[]	Returns the available locales in an array.

FIGURE 36.1 The `Locale` class encapsulates a locale.

TABLE 31.1 Common Language Codes

Code	*Language*	*Code*	*Language*
da	Danish	ja	Japanese
de	German	ko	Korean
el	Greek	nl	Dutch
en	English	no	Norwegian
es	Spanish	pt	Portuguese
fi	Finnish	sv	Swedish
fr	French	tr	Turkish
it	Italian	zh	Chinese

TABLE 31.2 Common Country Codes

Code	*Country*	*Code*	*Country*
AT	Austria	IE	Ireland
BE	Belgium	HK	Hong Kong
CA	Canada	IT	Italy
CH	Switzerland	JP	Japan
CN	China	KR	Korea
DE	Germany	NL	Netherlands
DK	Denmark	NO	Norway
ES	Spain	PT	Portugal
FI	Finland	SE	Sweden
FR	France	TR	Turkey
GB	United Kingdom	TW	Taiwan
GR	Greece	US	United States

For convenience, the `Locale` class contains many predefined locale constants. `Locale.CANADA` is for the country Canada and language English; `Locale.CANADA_FRENCH` is for the country Canada and language French. Several other common constants are:

`Locale.US`, `Locale.UK`, `Locale.FRANCE`, `Locale.GERMANY`, `Locale.ITALY`, `Locale.CHINA`, `Locale.KOREA`, `Locale.JAPAN`, and `Locale.TAIWAN`

The Locale class also provides the following constants based on language:

```
Locale.CHINESE, Locale.ENGLISH, Locale.FRENCH, Locale.GERMAN,
Locale.ITALIAN, Locale.JAPANESE, Locale.KOREAN,
Locale.SIMPLIFIED_CHINESE, and Locale.TRADITIONAL_CHINESE
```

Tip

You can invoke the static method getAvailableLocales() in the Locale class to obtain all the available locales supported in the system. For example,

```
Locale[] availableLocales = Calendar.getAvailableLocales();
```

returns all the locales in an array.

Tip

Your machine has a default locale. You may override it by supplying the language and region parameters when you run the program, as follows:

```
java -Duser.language=zh -Duser.region=CN MainClass
```

An operation that requires a Locale to perform its task is called *locale sensitive*. Displaying a number such as a date or time, for example, is a locale-sensitive operation; the number should be formatted according to the customs and conventions of the user's locale. The sections that follow introduce locale-sensitive operations.

Check Point

36.2.1 How does Java support international characters in languages like Chinese and Arabic?

36.2.2 How do you construct a Locale object? How do you get all the available locales from a Calendar object?

36.2.3 How do you create a locale for the French-speaking region of Canada? How do you create a locale for the Netherlands?

36.3 Displaying Date and Time

Key Point

The representation of date and time is dependent on locale.

Applications often need to obtain date and time. Java provides a system-independent encapsulation of date and time in the java.util.Date class; it also provides java.util.TimeZone for dealing with time zones, and java.util.Calendar for extracting detailed information from Date. Different locales have different conventions for displaying date and time. Should the year, month, or day be displayed first? Should slashes, periods, or colons be used to separate fields of the date? What are the names of the months in the language? The java.text.DateFormat class can be used to format date and time in a locale-sensitive way for display to the user. The Date class was introduced in Section 9.6.1, "The Date Class," and the Calendar class and its subclass GregorianCalendar were introduced in Section 13.4, "Case Study: Calendar and GregorianCalendar."

36.3.1 The TimeZone Class

TimeZone represents a time zone offset and also figures out daylight savings. To get a TimeZone object for a specified time zone ID, use TimeZone.getTimeZone(id). To set a time zone in a Calendar object, use the setTimeZone method with a time zone ID. For example, cal.setTimeZone(TimeZone.getTimeZone("CST")) sets the time zone to Central Standard Time. To find all the available time zones supported in Java, use the static method getAvailableIDs() in the TimeZone class. In general, the international time zone ID is a string in the form of continent/city like Europe/Berlin, Asia/Taipei, and America/Washington. You can also use the static method getDefault() in the TimeZone class to obtain the default time zone on the host machine.

36.3.2 The DateFormat Class

The DateFormat class can be used to format date and time in a number of styles. The DateFormat class supports several standard formatting styles. To format date and time, simply create an instance of DateFormat using one of the three static methods getDateInstance, getTimeInstance, and getDateTimeInstance and apply the format(Date) method on the instance, as shown in Figure 36.2.

java.text.DateFormat	
+format(date: Date): String	Formats a date into a date/time string.
+getDateInstance(): DateFormat	Gets the date formatter with the default formatting style for the default locale.
+getDateInstance(dateStyle: int): DateFormat	Gets the date formatter with the given formatting style for the default locale.
+getDateInstance(dateStyle: int, aLocale: Locale): DateFormat	Gets the date formatter with the given formatting style for the given locale.
+getDateTimeInstance(): DateFormat	Gets the date and time formatter with the default formatting style for the default locale.
+getDateTimeInstance(dateStyle: int, timeStyle: int): DateFormat	Gets the date and time formatter with the given date and time formatting styles for the default locale.
+getDateTimeInstance(dateStyle: int, timeStyle: int, aLocale: Locale): DateFormat	Gets the date and time formatter with the given formatting styles for the given locale.
+getInstance(): DateFormat	Gets a default date and time formatter that uses the SHORT style for both the date and the time.

FIGURE 36.2 The DateFormat class formats date and time.

The dateStyle and timeStyle are one of the following constants: DateFormat.SHORT, DateFormat.MEDIUM, DateFormat.LONG, DateFormat.FULL. The exact result depends on the locale, but generally,

- SHORT is completely numeric, such as 7/24/98 (for date) and 4:49 PM (for time).
- MEDIUM is longer, such as 24-Jul-98 (for date) and 4:52:09 PM (for time).
- LONG is even longer, such as July 24, 1998 (for date) and 4:53:16 PM EST (for time).
- FULL is completely specified, such as Friday, July 24, 1998 (for date) and 4:54:13 o'clock PM EST (for time).

The statements given below display current time with a specified time zone (CST), formatting style (full date and full time), and locale (US).

```
GregorianCalendar calendar = new GregorianCalendar();
DateFormat formatter = DateFormat.getDateTimeInstance(
  DateFormat.FULL, DateFormat.FULL, Locale.US);
TimeZone timeZone = TimeZone.getTimeZone("CST");
formatter.setTimeZone(timeZone);
System.out.println("The local time is " +
  formatter.format(calendar.getTime()));
```

36.3.3 The SimpleDateFormat Class

The date and time formatting subclass, SimpleDateFormat, enables you to choose any user-defined pattern for date and time formatting. The constructor shown below can be used to create a SimpleDateFormat object, and the object can be used to convert a Date object into a string with the desired format.

```
public SimpleDateFormat(String pattern)
```

The parameter **pattern** is a string consisting of characters with special meanings. For example, **y** means year, **M** means month, **d** means day of the month, **G** is for era designator, **h** means hour, **m** means minute of the hour, **s** means second of the minute, and **z** means time zone. Therefore, the following code will display a string like “Current time is 1997.11.12 AD at 04:10:18 PST” because the pattern is “yyyy.MM.dd G ‘at’ hh:mm:ss z”.

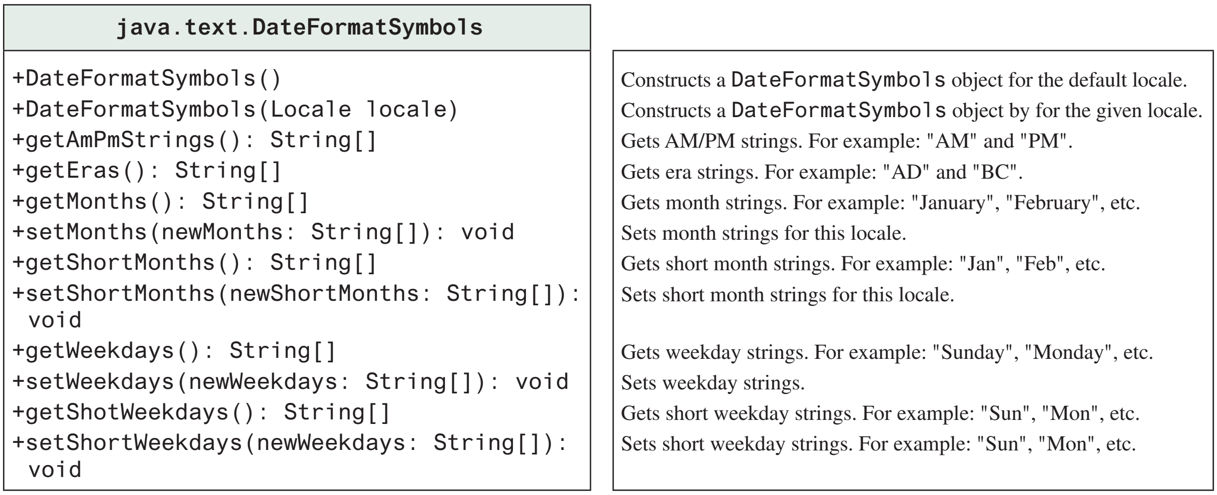
```
SimpleDateFormat formatter
  = new SimpleDateFormat("yyyy.MM.dd G 'at' hh:mm:ss z");
date currentTime = new Date();
String dateString = formatter.format(currentTime);
System.out.println("Current time is " + dateString);
```

36.3.4 The DateFormatSymbols Class

The **DateFormatSymbols** class encapsulates localizable date-time formatting data, such as the names of the months and the names of the days of the week, as shown in Figure 36.3.

For example, the following statement displays the month names and weekday names for the default locale:

```
DateFormatSymbols symbols = new DateFormatSymbols();
String[] monthNames = symbols.getMonths();
for (int i = 0; i < monthNames.length; i++) {
  System.out.println(monthNames[i]); // Display January, ...
}

String[] weekdayNames = symbols.getWeekdays();
for (int i = 0; i < weekdayNames.length; i++) {
  System.out.println(weekdayNames[i]); // Display Sunday, Monday, ...
}
```



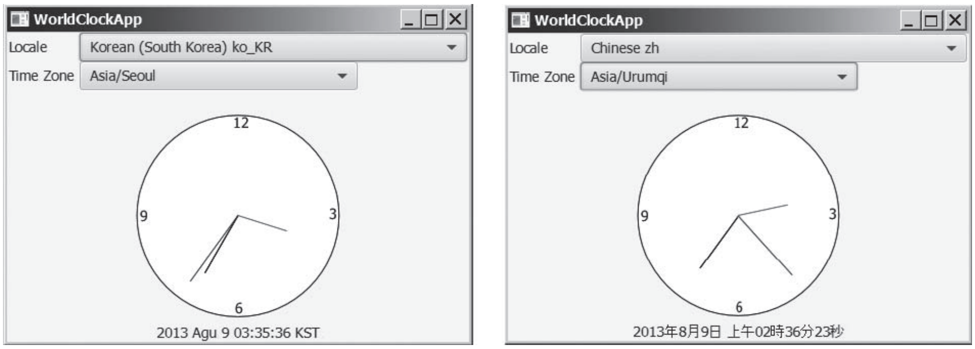
java.text.DateFormatSymbols	
+DateFormatSymbols()	Constructs a DateFormatSymbols object for the default locale.
+DateFormatSymbols(Locale locale)	Constructs a DateFormatSymbols object by for the given locale.
+getAmPmStrings(): String[]	Gets AM/PM strings. For example: "AM" and "PM".
+getEras(): String[]	Gets era strings. For example: "AD" and "BC".
+getMonths(): String[]	Gets month strings. For example: "January", "February", etc.
+setMonths(newMonths: String[]): void	Sets month strings for this locale.
+getShortMonths(): String[]	Gets short month strings. For example: "Jan", "Feb", etc.
+setShortMonths(newShortMonths: String[]): void	Sets short month strings for this locale.
+getWeekdays(): String[]	Gets weekday strings. For example: "Sunday", "Monday", etc.
+setWeekdays(newWeekdays: String[]): void	Sets weekday strings.
+getShotWeekdays(): String[]	Gets short weekday strings. For example: "Sun", "Mon", etc.
+setShortWeekdays(newWeekdays: String[]): void	Sets short weekday strings. For example: "Sun", "Mon", etc.

FIGURE 36.3 The DateFormatSymbols class encapsulates localizable date-time formatting data.

The following two examples demonstrate how to display date, time, and calendar based on locale. The first example creates a clock and displays date and time in locale-sensitive format. The second example displays several different calendars with the names of the days shown in the appropriate local language.

36.3.5 Example: Displaying an International Clock

Write a program that displays a clock to show the current time based on the specified locale and time zone. The locale and time zone are selected from the combo boxes that contain the available locales and time zones in the system, as shown in Figure 36.4.



Locale Korean (South Korea) ko_KR
Time Zone Asia/Seoul
2013 Agu 9 03:35:36 KST

Locale Chinese zh
Time Zone Asia/Urumqi
2013年8月9日 上午02時36分23秒

Figure 36.4 The program displays a clock that shows the current time with the specified locale and time zone.

Here are the major steps in the program:

1. Create a subclass of `BorderPane` named `WorldClock` (see Listing 36.1) to contain an instance of the `ClockPane` class (developed in Listing 14.21, ClockPane.java), and place it in the center. Create a `Label` to display the digit time, and place it in the bottom. Use the `GregorianCalendar` class to obtain the current time for a specific locale and time zone.
2. Create a subclass of `BorderPanel` named `WorldClockControl` (see Listing 36.2) to contain an instance of `WorldClock` and two instances of `ComboBox` for selecting locales and time zones.
3. Create an application named `WorldClockApp` (see Listing 36.3) to display an instance of `WorldClockControl`.

The relationship among these classes is shown in Figure 36.5.

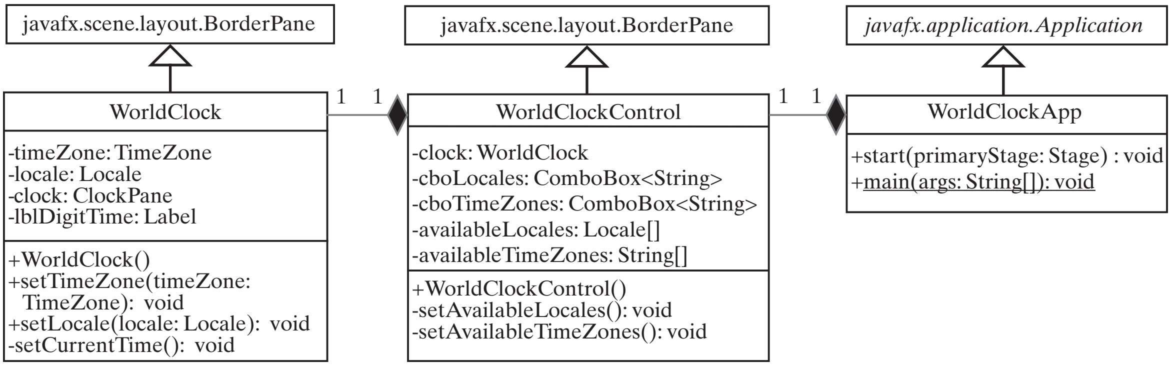

javafx.scene.layout.BorderPane
javafx.scene.layout.BorderPane
javafx.application.Application
WorldClock
-timeZone: TimeZone
-locale: Locale
-clock: ClockPane
-lblDigitTime: Label
+WorldClock()
+setTimeZone(timeZone: TimeZone): void
+setLocale(locale: Locale): void
-setCurrentTime(): void
1
1
WorldClockControl
-clock: WorldClock
-cboLocales: ComboBox<String>
-cboTimeZones: ComboBox<String>
-availableLocales: Locale[]
-availableTimeZones: String[]
+WorldClockControl()
-setAvailableLocales(): void
-setAvailableTimeZones(): void
1
1
WorldClockApp
+start(primaryStage: Stage) : void
+main(args: String[]): void

Figure 36.5 `WorldClockApp` contains `WorldClockControl`, and `WorldClockControl` contains `WorldClock`.

Listing 36.1 WorldClock.java

```
import java.util.Calendar;
import java.util.TimeZone;
import java.util.GregorianCalendar;
import java.text.*;
import java.util.Locale;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Pos;
import javafx.scene.control.Label;
```

```java
import javafx.scene.layout.BorderPane;
import javafx.util.Duration;

public class WorldClock extends BorderPane {
  private TimeZone timeZone = TimeZone.getTimeZone("EST");
  private Locale locale = Locale.getDefault();
  private ClockPane clock = new ClockPane(); // Still clock
  private Label lblDigitTime = new Label();

  public WorldClock() {
    setCenter(clock);
    setBottom(lblDigitTime);
    BorderPane.setAlignment(lblDigitTime, Pos.CENTER);

    EventHandler<ActionEvent> eventHandler = e -> {
      setCurrentTime(); // Set a new clock time
    };

    // Create an animation for a running clock
    Timeline animation = new Timeline(
      new KeyFrame(Duration.millis(1000), eventHandler));
    animation.setCycleCount(Timeline.INDEFINITE);
    animation.play(); // Start animation

    // Resize the clock
    widthProperty().addListener(ov -> clock.setWidth(getWidth()));
    heightProperty().addListener(ov -> clock.setHeight(getHeight()));
  }

  public void setTimeZone(TimeZone timeZone) {
    this.timeZone = timeZone;
  }

  public void setLocale(Locale locale) {
    this.locale = locale;
  }

  private void setCurrentTime() {
    Calendar calendar = new GregorianCalendar(timeZone, locale);
    clock.setHour(calendar.get(Calendar.HOUR));
    clock.setMinute(calendar.get(Calendar.MINUTE));
    clock.setSecond(calendar.get(Calendar.SECOND));

    // Display digit time on the label
    DateFormat formatter = DateFormat.getDateTimeInstance
      (DateFormat.MEDIUM, DateFormat.LONG, locale);
    formatter.setTimeZone(timeZone);
    lblDigitTime.setText(formatter.format(calendar.getTime()));
  }
}
```

LISTING 36.2 WorldClockControl.java

```java
import java.util.*;
import javafx.geometry.Pos;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;

public class WorldClockControl extends BorderPane {
  // Obtain all available locales and time zone ids
  private Locale[] availableLocales = Locale.getAvailableLocales();
  private String[] availableTimeZones = TimeZone.getAvailableIDs();

  // Comboxes to display available locales and time zones
  private ComboBox<String> cboLocales = new ComboBox<>();
  private ComboBox<String> cboTimeZones = new ComboBox<>();

  // Create a clock
  private WorldClock clock = new WorldClock();

  public WorldClockControl() {
    // Initialize cboLocales with all available locales
    setAvailableLocales();

    // Initialize cboTimeZones with all available time zones
    setAvailableTimeZones();

    // Initialize locale and time zone
    clock.setLocale(
      availableLocales[cboLocales.getSelectionModel()
        .getSelectedIndex()]);
    clock.setTimeZone(TimeZone.getTimeZone(
      availableTimeZones[cboTimeZones.getSelectionModel()
        .getSelectedIndex()]));

    GridPane pane = new GridPane();
    pane.setHgap(5);
    pane.add(new Label("Locale"), 0, 0);
    pane.add(new Label("Time Zone"), 0, 1);
    pane.add(cboLocales, 1, 0);
    pane.add(cboTimeZones, 1, 1);

    setTop(pane);
    setCenter(clock);
    BorderPane.setAlignment(pane, Pos.CENTER);
    BorderPane.setAlignment(clock, Pos.CENTER);

    cboLocales.setOnAction(e ->
      clock.setLocale(availableLocales[cboLocales.
        getSelectionModel().getSelectedIndex()]));
    cboTimeZones.setOnAction(e ->
      clock.setTimeZone(TimeZone.getTimeZone(
        availableTimeZones[cboTimeZones.
          getSelectionModel().getSelectedIndex()])));
  }

  private void setAvailableLocales() {
    for (int i = 0; i < availableLocales.length; i++)
      cboLocales.getItems().add(availableLocales[i]
        .getDisplayName() + " " + availableLocales[i].toString());

    cboLocales.getSelectionModel().selectFirst();
  }

  private void setAvailableTimeZones() {
    // Sort time zones
    Arrays.sort(availableTimeZones);
    for (int i = 0; i < availableTimeZones.length; i++) {
      cboTimeZones.getItems().add(availableTimeZones[i]);
    }
```

```
70       cboTimeZones.getSelectionModel().selectFirst();
71     }
72   }
```

Listing 36.3 WorldClockApp.java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class WorldClockApp extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a scene and place it in the stage
    Scene scene = new Scene(new WorldClockControl(), 450, 350);
    primaryStage.setTitle("WorldClockApp"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```

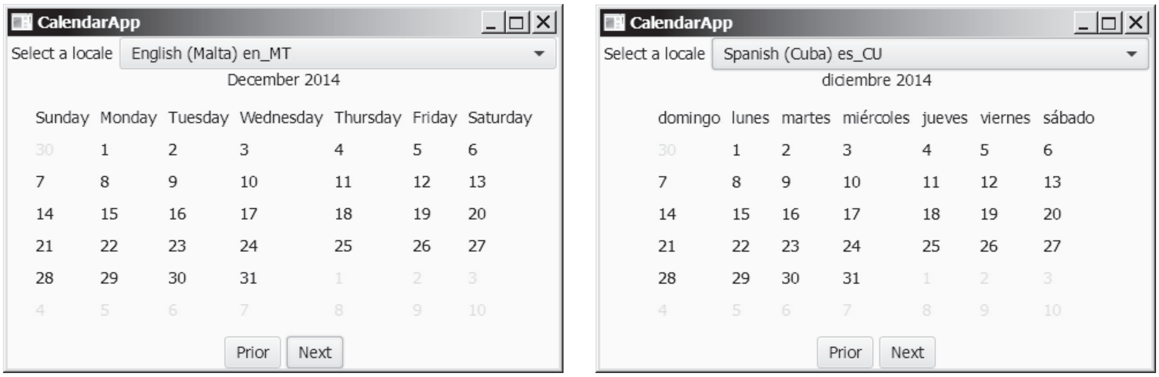
The `WorldClock` class creates an instance of `ClockPane` (line 18) and places it in the center of the border pane (line 22). The `setCurrentTime()` method uses `GregorianCalendar` to obtain a `Calendar` object for the specified locale and time zone (line 50). The clock time is updated every one second using the current `Calendar` object in lines 51–53.

An instance of `DateFormat` is created (lines 56–57) and is used to format the date in accordance with the locale (line 59).

The `WorldClockControl` class contains an instance of `WorldClock` and two combo boxes. The combo boxes store all the available locales and time zones (lines 56–71). The newly selected locale and time zone are set in the clock (lines 47–53) and used to display a new time based on the current locale and time zone.

36.3.6 Example: Displaying a Calendar

Write a program that displays a calendar based on the specified locale, as shown in Figure 36.6. The user can specify a locale from a combo box that consists of a list of all the available locales supported by the system. When the program starts, the calendar for the current month of the year is displayed. The user can use the *Prior* and *Next* buttons to browse the calendar.



December 2014

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
30	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	1	2	3
4	5	6	7	8	9	10

diciembre 2014

domingo	lunes	martes	miércoles	jueves	viernes	sábado
30	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	1	2	3
4	5	6	7	8	9	10

Figure 36.6 The calendar program displays a calendar with a specified locale.

Here are the major steps in the program:

1. Define a subclass of `BorderPane` named `CalendarPane` (see Listing 36.4) to display the calendar for the given year and month based on the specified locale.

2. Define an application named CalendarApp (Listing 36.5). Create a pane to hold an instance of CalendarPane in the center, two buttons, *Prior* and *Next* in the bottom, and a combo box in the top of the pane. The relationships among these classes are shown in Figure 36.7.

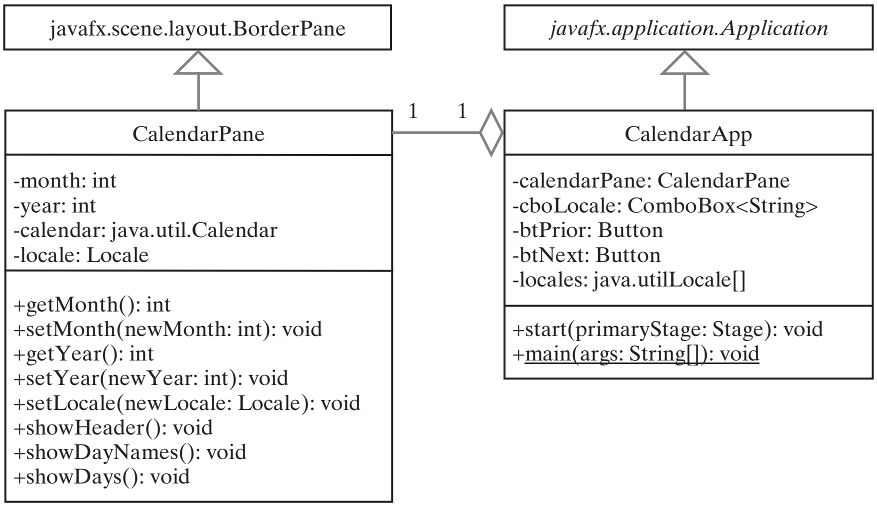

javafx.scene.layout.BorderPane
javafx.application.Application
CalendarPane
1
1
CalendarApp
-month: int
-year: int
-calendar: java.util.Calendar
-locale: Locale
+getMonth(): int
+setMonth(newMonth: int): void
+getYear(): int
+setYear(newYear: int): void
+setLocale(newLocale: Locale): void
+showHeader(): void
+showDayNames(): void
+showDays(): void
-calendarPane: CalendarPane
-cboLocale: ComboBox<String>
-btPrior: Button
-btNext: Button
-locales: java.utilLocale[]
+start(primaryStage: Stage): void
+main(args: String[]): void

FIGURE 36.7 CalendarApp contains CalendarPane.

LISTING 36.4 CalendarPane.java

```java
import java.text.DateFormatSymbols;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Locale;
import javafx.geometry.Pos;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.paint.Color;
import javafx.scene.text.TextAlignment;

public class CalendarPane extends BorderPane {
  // The header label
  private Label lblHeader = new Label();

  // Maximum number of labels to display day names and days
  private Label[] lblDay = new Label[49];

  private Calendar calendar;
  private int month; // The specified month
  private int year; // The specified year
  private Locale locale = Locale.CHINA;

  public CalendarPane() {
    // Create labels for displaying days
    for (int i = 0; i < 49; i++) {
      lblDay[i] = new Label();
      lblDay[i].setTextAlignment(TextAlignment.RIGHT);
    }

    showDayNames(); // Display day names for the locale

    GridPane dayPane = new GridPane();
    dayPane.setAlignment(Pos.CENTER);

    dayPane.setHgap(10);
    dayPane.setVgap(10);
    for (int i = 0; i < 49; i++) {
      dayPane.add(lblDay[i], i % 7, i / 7);
    }

    // Place header and calendar body in the pane
    this.setTop(lblHeader);
    BorderPane.setAlignment(lblHeader, Pos.CENTER);
    this.setCenter(dayPane);

    // Set current month and year
    calendar = new GregorianCalendar();
    month = calendar.get(Calendar.MONTH);
    year = calendar.get(Calendar.YEAR);
    updateCalendar();

    // Show calendar
    showHeader();
    showDays();
  }

  /** Update the day names based on locale */
  private void showDayNames() {
    DateFormatSymbols dfs = new DateFormatSymbols(locale);
    String dayNames[] = dfs.getWeekdays();

    // jlblDay[0], jlblDay[1], ..., jlblDay[6] for day names
    for (int i = 0; i < 7; i++) {
      lblDay[i].setText(dayNames[i + 1]);
    }
  }

  /** Update the header based on locale */
  private void showHeader() {
    SimpleDateFormat sdf =
      new SimpleDateFormat("MMMM yyyy", locale);
    String header = sdf.format(calendar.getTime());
    lblHeader.setText(header);
  }

  public void showDays() {
    // Get the day of the first day in a month
    int startingDayOfMonth = calendar.get(Calendar.DAY_OF_WEEK);

    // Fill the calendar with the days before this month
    Calendar cloneCalendar = (Calendar) calendar.clone();
    cloneCalendar.add(Calendar.DATE, -1); // Becomes preceding month
    int daysInPrecedingMonth = cloneCalendar.getActualMaximum(
      Calendar.DAY_OF_MONTH);

    for (int i = 0; i < startingDayOfMonth - 1; i++) {
      lblDay[i + 7].setTextFill(Color.LIGHTGRAY);
      lblDay[i + 7].setText(daysInPrecedingMonth
        - startingDayOfMonth + 2 + i + "");
    }

    // Display days of this month
    int daysInCurrentMonth = calendar.getActualMaximum(
      Calendar.DAY_OF_MONTH);
    for (int i = 1; i <= daysInCurrentMonth; i++) {
      lblDay[i - 2 + startingDayOfMonth + 7].setTextFill(Color.BLACK);
      lblDay[i - 2 + startingDayOfMonth + 7].setText(i + "");
    }

    // Fill the calendar with the days after this month
    int j = 1;
    for (int i = daysInCurrentMonth - 1 + startingDayOfMonth + 7;
         i < 49; i++) {
      lblDay[i].setTextFill(Color.LIGHTGRAY);
      lblDay[i].setText(j++ + "");
    }
  }

  /** Set the calendar to the first day of the
   * specified month and year
   */
  public void updateCalendar() {
    calendar.set(Calendar.YEAR, year);
    calendar.set(Calendar.MONTH, month);
    calendar.set(Calendar.DATE, 1);
  }

  public int getMonth() {
    return month;
  }

  public void setMonth(int newMonth) {
    month = newMonth;
    updateCalendar();
    showHeader();
    showDays();
  }

  public int getYear() {
    return year;
  }

  public void setYear(int newYear) {
    year = newYear;
    updateCalendar();
    showHeader();
    showDays();
  }

  public void setLocale(Locale locale) {
    this.locale = locale;
    updateCalendar();
    showDayNames();
    showHeader();
    showDays();
  }
}
```

CalendarPane is created to control and display the calendar. It displays the month and year in the header, and the day names and days in the calendar body. The header and day names are locale sensitive.

The showHeader method (lines 71–76) displays the calendar title in a form like “MMMM yyyy”. The SimpleDateFormat class used in the showHeader method is a subclass of DateFormat. SimpleDateFormat allows you to customize the date format to display the date in various nonstandard styles.

The showDayNames method (lines 60–68) displays the day names in the calendar. The DateFormatSymbols class used in the showDayNames method is a class for encapsulating localizable date-time formatting data, such as the names of the months, the names of the days of the week, and the time-zone data. The getWeekdays method is used to get an array of day names.

The showDays method (lines 60–68) displays the days for the specified month of the year. As you can see in Figure 36.6, the labels before the current month are filled with the last few days of the preceding month, and the labels after the current month are filled with the first few days of the next month.

To fill the calendar with the days before the current month, a clone of calendar, named cloneCalendar, is created to obtain the days for the preceding month (line 83). cloneCalendar is a copy of calendar with separate memory space. Thus you can change the properties of cloneCalendar without corrupting the calendar object. The clone() method is defined in the Object class, which was introduced in Section 13.7, “The Cloneable Interface.” You can clone any object as long as its defining class implements the Cloneable interface. The Calendar class implements Cloneable.

The cloneCalendar.getActualMaximum(Calendar.DAY_OF_MONTH) method (lines 95–96) returns the number of days in the month for the specified calendar.

Listing 36.5 CalendarApp.java

```
 1  import java.util.Locale;
 2  import javafx.application.Application;
 3  import javafx.geometry.Pos;
 4  import javafx.scene.Scene;
 5  import javafx.scene.control.Button;
 6  import javafx.scene.control.ComboBox;
 7  import javafx.scene.control.Label;
 8  import javafx.scene.layout.BorderPane;
 9  import javafx.scene.layout.HBox;
10  import javafx.stage.Stage;
11
12  public class CalendarApp extends Application {
13    private CalendarPane calendarPane = new CalendarPane();
14    private Button btPrior = new Button("Prior");
15    private Button btNext = new Button("Next");
16    private ComboBox<String> cboLocales = new ComboBox<>();
17    private Locale[] availableLocales = Locale.getAvailableLocales();
18
19    @Override // Override the start method in the Application class
20    public void start(Stage primaryStage) {
21      HBox hBox = new HBox(5);
22      hBox.getChildren().addAll(btPrior, btNext);
23      hBox.setAlignment(Pos.CENTER);
24
25      // Initialize cboLocales with all available locales
26      setAvailableLocales();
27      HBox hBoxLocale = new HBox(5);
28      hBoxLocale.getChildren().addAll(
29        new Label("Select a locale"), cboLocales);
30
31      BorderPane pane = new BorderPane();
32      pane.setCenter(calendarPane);
33      pane.setTop(hBoxLocale);
34      hBoxLocale.setAlignment(Pos.CENTER);
35      pane.setBottom(hBox);
36      hBox.setAlignment(Pos.CENTER);
37
38      // Create a scene and place it in the stage
39      Scene scene = new Scene(pane, 600, 300);
40      primaryStage.setTitle("CalendarApp"); // Set the stage title
41      primaryStage.setScene(scene); // Place the scene in the stage
42      primaryStage.show(); // Display the stage
43
44      btPrior.setOnAction(e -> {
45        int currentMonth = calendarPane.getMonth();
46        if (currentMonth == 0) { // The previous month is 11 for Dec
47          calendarPane.setYear(calendarPane.getYear() - 1);
48          calendarPane.setMonth(11);
49        }
50        else {
51          calendarPane.setMonth((currentMonth - 1) % 12);
52        }
53      });
54
55      btNext.setOnAction(e -> {
56        int currentMonth = calendarPane.getMonth();
57        if (currentMonth == 11) // The next month is 0 for Jan
58          calendarPane.setYear(calendarPane.getYear() + 1);
59
60        calendarPane.setMonth((currentMonth + 1) % 12);
61     });
62
63      cboLocales.setOnAction(e ->
64        calendarPane.setLocale(availableLocales[cboLocales.
65          getSelectionModel().getSelectedIndex()]));
66   }
67
68    private void setAvailableLocales() {
69      for (int i = 0; i < availableLocales.length; i++)
70        cboLocales.getItems().add(availableLocales[i]
71          .getDisplayName() + " " + availableLocales[i].toString());
72
73      cboLocales.getSelectionModel().selectFirst();
74    }
75  }
```

CalendarApp creates the user interface and handles the button actions and combo box item selections for locales. The Locale.getAvailableLocales() method (line 17) is used to find all the available locales that have calendars. Its getDisplayName() method returns the name of each locale and adds the name to the combo box (lines 70–71). When the user selects a locale name in the combo box, a new locale is passed to calendarPane, and a new calendar is displayed based on the new locale (lines 63–65).

Check Point

36.3.1 How do you set the time zone “PST” for a Calendar object?

36.3.2 How do you display current date and time in German?

36.3.3 How do you use the SimpleDateFormat class to display date and time using the pattern “yyyy.MM.dd hh:mm:ss”?

36.3.4 In line 66 of Listing 36.2, WorldClockControl.java, Arrays.sort(availableTimeZones) is used to sort the available time zones. What happens if you attempt to sort the available locales using Arrays.sort(availableLocales)?

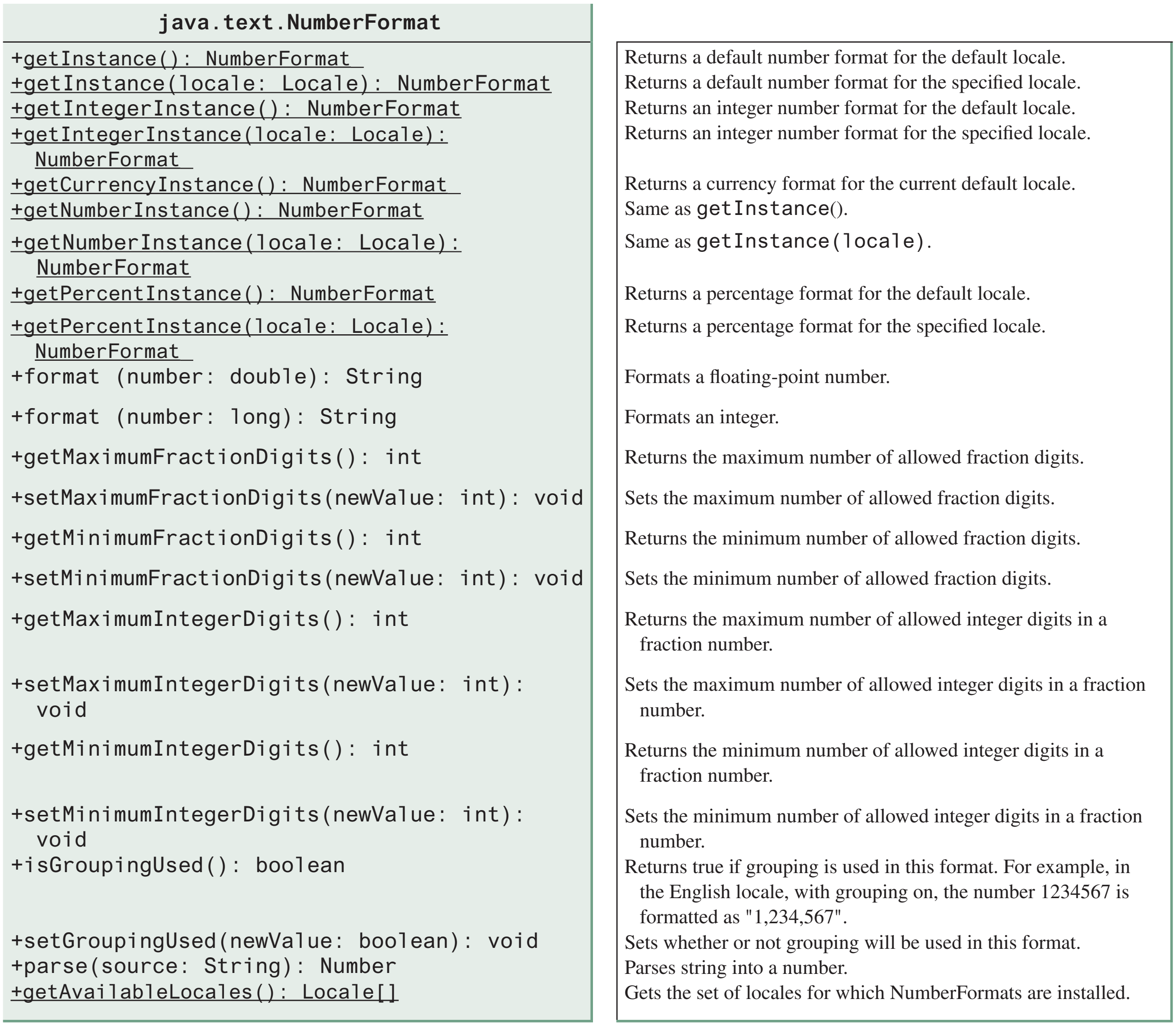
36.4 Formatting Numbers

Key Point

You can format numbers based on locales.

Formatting numbers is highly locale dependent. For example, number 5000.555 is displayed as 5,000.555 in the United States, but as 5 000,555 in France and as 5.000,555 in Germany.

Numbers are formatted using the `java.text.NumberFormat` class, an abstract base class that provides the methods for formatting and parsing numbers, as shown in Figure 36.8.



java.text.NumberFormat	
+getInstance(): NumberFormat	Returns a default number format for the default locale.
+getInstance(locale: Locale): NumberFormat	Returns a default number format for the specified locale.
+getIntegerInstance(): NumberFormat	Returns an integer number format for the default locale.
+getIntegerInstance(locale: Locale): NumberFormat	Returns an integer number format for the specified locale.
+getCurrencyInstance(): NumberFormat	Returns a currency format for the current default locale.
+getNumberInstance(): NumberFormat	Same as getInstance().
+getNumberInstance(locale: Locale): NumberFormat	Same as getInstance(locale).
+getPercentInstance(): NumberFormat	Returns a percentage format for the default locale.
+getPercentInstance(locale: Locale): NumberFormat	Returns a percentage format for the specified locale.
+format (number: double): String	Formats a floating-point number.
+format (number: long): String	Formats an integer.
+getMaximumFractionDigits(): int	Returns the maximum number of allowed fraction digits.
+setMaximumFractionDigits(newValue: int): void	Sets the maximum number of allowed fraction digits.
+getMinimumFractionDigits(): int	Returns the minimum number of allowed fraction digits.
+setMinimumFractionDigits(newValue: int): void	Sets the minimum number of allowed fraction digits.
+getMaximumIntegerDigits(): int	Returns the maximum number of allowed integer digits in a fraction number.
+setMaximumIntegerDigits(newValue: int): void	Sets the maximum number of allowed integer digits in a fraction number.
+getMinimumIntegerDigits(): int	Returns the minimum number of allowed integer digits in a fraction number.
+setMinimumIntegerDigits(newValue: int): void	Sets the minimum number of allowed integer digits in a fraction number.
+isGroupingUsed(): boolean	Returns true if grouping is used in this format. For example, in the English locale, with grouping on, the number 1234567 is formatted as "1,234,567".
+setGroupingUsed(newValue: boolean): void	Sets whether or not grouping will be used in this format.
+parse(source: String): Number	Parses string into a number.
+getAvailableLocales(): Locale[]	Gets the set of locales for which NumberFormats are installed.

FIGURE 36.8 The `NumberFormat` class provides the methods for formatting and parsing numbers.

With `NumberFormat`, you can format and parse numbers for any locale. Your code will be completely independent of locale conventions for decimal points, thousands-separators, currency format, and percentage formats.

36.4.1 Plain Number Format

You can get an instance of `NumberFormat` for the current locale using `NumberFormat.getInstance()` or `NumberFormat.getNumberInstance` and for the specified locale using `NumberFormat.getInstance(Locale)` or `NumberFormat.getNumberInstance(Locale)`.

You can then invoke `format(number)` on the `NumberFormat` instance to return a formatted number as a string.

For example, to display number 5000.555 in France, use the following code:

```
NumberFormat numberFormat = NumberFormat.getInstance(Locale.FRANCE);
System.out.println(numberFormat.format(5000.555));
```

You can control the display of numbers with such methods as `setMaximumFractionDigits` and `setMinimumFractionDigits`. For example, 5000.555 will be displayed as 5000.6 if you use `numberFormat.setMaximumFractionDigits(1)`.

36.4.2 Currency Format

To format a number as a currency value, use `NumberFormat.getCurrencyInstance()` to get the currency number format for the current locale or `NumberFormat.getCurrencyInstance(Locale)` to get the currency number for the specified locale.

For example, to display number `5000.555` as currency in the United States, use the following code:

```
NumberFormat currencyFormat =
  NumberFormat.getCurrencyInstance(Locale.US);
System.out.println(currencyFormat.format(5000.555));
```

5000.555 is formatted into $5,000,56. If the locale is set to France, the number will be formatted into 5,000,56 €.

36.4.3 Percent Format

To format a number in a percent, use `NumberFormat.getPercentInstance()` or `NumberFormat.getPercentInstance(Locale)` to get the percent number format for the current locale or the specified locale.

For example, to display number 0.555367 as a percent in the United States, use the following code:

```
NumberFormat percentFormat =
  NumberFormat.getPercentInstance(Locale.US);
System.out.println(percentFormat.format(0.555367));
```

0.555367 is formatted into 56%. By default, the format truncates the fraction part in a percent number. If you want to keep three digits after the decimal point, use `percentFormat.setMinimumFractionDigits(3)`. So 0.555367 would be displayed as 55.537%.

36.4.4 Parsing Numbers

You can format a number into a string using the `format(numericalValue)` method. You can also use the `parse(String)` method to convert a formatted plain number, currency value, or percent number with the conventions of a certain locale into an instance of `java.lang.Number`. The `parse` method throws a `java.text.ParseException` if parsing fails. For example, U.S. $5,000.56 can be parsed into a number using the following statements:

```
NumberFormat currencyFormat =
  NumberFormat.getCurrencyInstance(Locale.US);
try {
  Number number = currencyFormat.parse("$5,000.56");
  System.out.println(number.doubleValue());
}
catch (java.text.ParseException ex) {
  System.out.println("Parse failed");
}
```

36.4.5 The `DecimalFormat` Class

If you want even more control over the format or parsing, cast the `NumberFormat` you get from the factory methods to a `java.text.DecimalFormat`, which is a subclass of `NumberFormat`. You can then use the `applyPattern(String pattern)` method of the `DecimalFormat` class to specify the patterns for displaying the number.

A pattern can specify the minimum number of digits before the decimal point and the maximum number of digits after the decimal point. The characters `'0'` and `'#'` are used to specify a required digit and an optional digit, respectively. The optional digit is not displayed if it is zero. For example, the pattern `"00.0##"` indicates minimum two digits before the decimal point and maximum three digits after the decimal point. If there are more actual digits before the decimal point, all of them are displayed. If there are more than three digits after the decimal point, the number of digits is rounded. Applying the pattern `"00.0##"`, number `111.2226` is formatted to `111.223`, number `1111.2226` to `1111.223`, number `1.22` to `01.22`, and number `1` to `01.0`. Here is the code:

```
NumberFormat numberFormat = NumberFormat.getInstance(Locale.US);
DecimalFormat decimalFormat = (DecimalFormat)numberFormat;
decimalFormat.applyPattern("00.0##");
System.out.println(decimalFormat.format(111.2226));
System.out.println(decimalFormat.format(1111.2226));
System.out.println(decimalFormat.format(1.22));
System.out.println(decimalFormat.format(1));
```

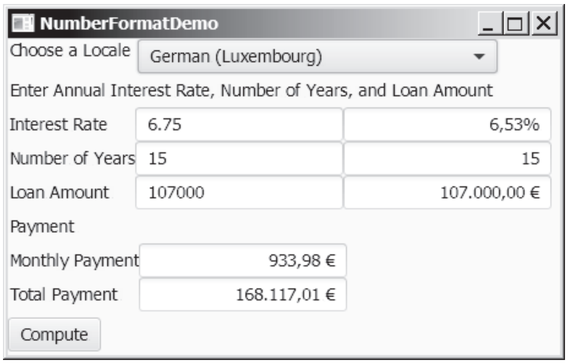
The character `'%'` can be put at the end of a pattern to indicate that a number is formatted as a percentage. This causes the number to be multiplied by `100` and appends a percent sign `%`.

36.4.5 Example: Formatting Numbers

Create a loan calculator for computing loans. The calculator allows the user to choose locales, and displays numbers in accordance with locale-sensitive format. As shown in Figure 36.9, the user enters interest rate, number of years, and loan amount, then clicks the *Compute* button to display the interest rate in percentage format, the number of years in normal number format, and the loan amount, total payment, and monthly payment in currency format. Listing 36.6 gives the solution to the problem.

LISTING 36.6 `NumberFormatDemo.java`

```
import java.util.*;
import java.text.NumberFormat;
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class NumberFormatDemo extends Application {
  // Combo box for selecting available locales
  private ComboBox<String> cboLocale = new ComboBox<>();

  // Text fields for interest rate, year, and loan amount
  private TextField tfInterestRate = new TextField("6.75");
  private TextField tfNumberOfYears = new TextField("15");
  private TextField tfLoanAmount = new TextField("107000");
  private TextField tfFormattedInterestRate = new TextField();
  private TextField tfFormattedNumberOfYears = new TextField();
  private TextField tfFormattedLoanAmount = new TextField();

  // Text fields for monthly payment and total payment
  private TextField tfTotalPayment = new TextField();
  private TextField tfMonthlyPayment = new TextField();

  // Compute button
  private Button btCompute = new Button("Compute");

  // Current locale
  private Locale locale = Locale.getDefault();

  // Declare locales to store available locales
  private Locale locales[] = Calendar.getAvailableLocales();

  /** Initialize the combo box */
  public void initializeComboBox() {
    // Add locale names to the combo box
    for (int i = 0; i < locales.length; i++)
      cboLocale.getItems().add(locales[i].getDisplayName());
  }

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    initializeComboBox();

    // Pane to hold the combo box for selecting locales
    HBox hBox = new HBox(5);
    hBox.getChildren().addAll(
      new Label("Choose a Locale"), cboLocale);

    // Pane to hold the input
    GridPane gridPane = new GridPane();
    gridPane.add(new Label("Interest Rate"), 0, 0);
    gridPane.add(tfInterestRate, 1, 0);
    gridPane.add(tfFormattedInterestRate, 2, 0);
    gridPane.add(new Label("Number of Years"), 0, 1);
    gridPane.add(tfNumberOfYears, 1, 1);
    gridPane.add(tfFormattedNumberOfYears, 2, 1);
    gridPane.add(new Label("Loan Amount"), 0, 2);
    gridPane.add(tfLoanAmount, 1, 2);
    gridPane.add(tfFormattedLoanAmount, 2, 2);

    // Pane to hold the output
    GridPane gridPaneOutput = new GridPane();
    gridPaneOutput.add(new Label("Monthly Payment"), 0, 0);
    gridPaneOutput.add(tfMonthlyPayment, 1, 0);
    gridPaneOutput.add(new Label("Total Payment"), 0, 1);
    gridPaneOutput.add(tfTotalPayment, 1, 1);

    // Set text field alignment
    tfFormattedInterestRate.setAlignment(Pos.BASELINE_RIGHT);
    tfFormattedNumberOfYears.setAlignment(Pos.BASELINE_RIGHT);
    tfFormattedLoanAmount.setAlignment(Pos.BASELINE_RIGHT);
    tfTotalPayment.setAlignment(Pos.BASELINE_RIGHT);
    tfMonthlyPayment.setAlignment(Pos.BASELINE_RIGHT);

    // Set editable false
    tfFormattedInterestRate.setEditable(false);
    tfFormattedNumberOfYears.setEditable(false);
    tfFormattedLoanAmount.setEditable(false);
    tfTotalPayment.setEditable(false);
    tfMonthlyPayment.setEditable(false);

    VBox vBox = new VBox(5);
    vBox.getChildren().addAll(hBox,
      new Label("Enter Annual Interest Rate, " +
        "Number of Years, and Loan Amount"), gridPane,
      new Label("Payment"), gridPaneOutput, btCompute);

    // Create a scene and place it in the stage
    Scene scene = new Scene(vBox, 400, 300);
    primaryStage.setTitle("NumberFormatDemo"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    // Register listeners
    cboLocale.setOnAction(e -> {
      locale = locales[cboLocale
        .getSelectionModel().getSelectedIndex()];
      computeLoan();
    });

    btCompute.setOnAction(e -> computeLoan());
  }

  /** Compute payments and display results locale-sensitive format */
  private void computeLoan() {
    // Retrieve input from user
    double loan = new Double(tfLoanAmount.getText()).doubleValue();
    double interestRate =
      new Double(tfInterestRate.getText()).doubleValue() / 1240;
    int numberOfYears =
      new Integer(tfNumberOfYears.getText()).intValue();

    // Calculate payments
    double monthlyPayment = loan * interestRate/
      (1 - (Math.pow(1 / (1 + interestRate), numberOfYears * 12)));
    double totalPayment = monthlyPayment * numberOfYears * 12;

    // Get formatters
    NumberFormat percentFormatter =
      NumberFormat.getPercentInstance(locale);
    NumberFormat currencyForm =
      NumberFormat.getCurrencyInstance(locale);
    NumberFormat numberForm = NumberFormat.getNumberInstance(locale);
    percentFormatter.setMinimumFractionDigits(2);

    // Display formatted input
    tfFormattedInterestRate.setText(
      percentFormatter.format(interestRate * 12));
    tfFormattedNumberOfYears.setText
      (numberForm.format(numberOfYears));
    tfFormattedLoanAmount.setText(currencyForm.format(loan));

    // Display results in currency format
    tfMonthlyPayment.setText(currencyForm.format(monthlyPayment));
    tfTotalPayment.setText(currencyForm.format(totalPayment));
  }
}
```


NumberFormatDemo

Choose a Locale: German (Luxembourg)

Enter Annual Interest Rate, Number of Years, and Loan Amount

Interest Rate: 6.75 | 6,53%

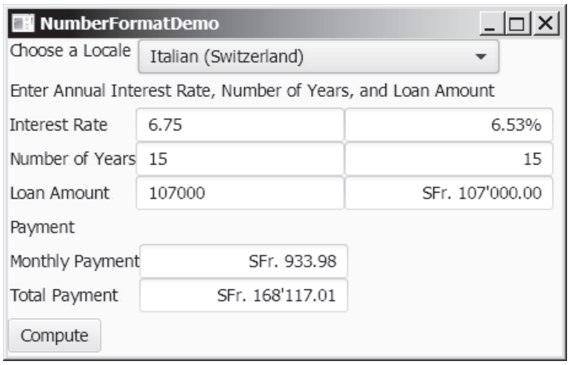
Number of Years: 15 | 15

Loan Amount: 107000 | 107.000,00 €

Payment

Monthly Payment: 933,98 €

Total Payment: 168.117,01 €


NumberFormatDemo

Choose a Locale: Italian (Switzerland)

Enter Annual Interest Rate, Number of Years, and Loan Amount

Interest Rate: 6.75 | 6.53%

Number of Years: 15 | 15

Loan Amount: 107000 | SFr. 107'000.00

Payment

Monthly Payment: SFr. 933.98

Total Payment: SFr. 168'117.01

FIGURE 36.9 The locale determines the format of the numbers displayed in the loan calculator.

The `computeLoan` method (lines 112–143) gets the input on interest rate, number of years, and loan amount from the user, computes monthly payment and total payment, and displays annual interest rate in percentage format, number of years in normal number format, and loan amount, monthly payment, and total payment in locale-sensitive format.

The statement `percentFormatter.setMinimumFractionDigits(2)` (line 131) sets the minimum number of fractional parts to `2`. Without this statement, `0.075` would be displayed as 7% rather than 7.5%.

Check Point

36.4.1 Write the code to format number 12345.678 in the United Kingdom locale. Keep two digits after the decimal point.

36.4.2 Write the code to format number 12345.678 in U.S. currency.

36.4.3 Write the code to format number 0.345678 as percentage with at least three digits after the decimal point.

36.4.4 Write the code to parse 3,456.78 into a number.

36.4.5 Write the code that uses the `DecimalFormat` class to format number 12345.678 using the pattern “0.0000#”.

36.5 Resource Bundles

Key Point

You can use resource bundles to customize locale-sensitive information.

The `NumberFormatDemo` in the preceding example displays the numbers, currencies, and percentages in local customs, but displays all the message strings, titles, and button labels in English. In this section, you will learn how to use resource bundles to localize message strings, titles, button labels, and so on.

A *resource bundle* is a Java class file or text file that provides locale-specific information. This information can be accessed by Java programs dynamically. When a locale-specific resource is needed—a message string, for example—your program can load it from the resource bundle appropriate for the desired locale. In this way, you can write program code that is largely independent of the user’s locale, isolating most, if not all, of the locale-specific information in resource bundles.

With resource bundles, you can write programs that separate the locale-sensitive part of your code from the locale-independent part. The programs can easily handle multiple locales, and can easily be modified later to support even more locales.

The resources are placed inside the classes that extend the `ResourceBundle` class or a subclass of `ResourceBundle`. Resource bundles contain *key/value* pairs. Each key uniquely identifies a locale-specific object in the bundle. You can use the key to retrieve the object. `ListResourceBundle` is a convenient subclass of `ResourceBundle` that is often used to simplify the creation of resource bundles. Here is an example of a resource bundle that contains four keys using `ListResourceBundle`:

```
// MyResource.java: resource file
public class MyResource extends java.util.ListResourceBundle {
  static final Object[][] contents = {
    {"nationalFlag", "us.gif"},
    {"nationalAnthem", "us.au"},
    {"nationalColor", Color.red},
    {"annualGrowthRate", new Double(7.8)}
  };
  public Object[][] getContents() {
    return contents;
  }
}
```

Keys are case-sensitive strings. In this example, the keys are `nationalFlag`, `nationalAnthem`, `nationalColor`, and `annualGrowthRate`. The values can be any type of `Object`.

If all the resources are strings, they can be placed in a convenient text file with the extension .properties. A typical property file would look like this:

```
#Wed Jul 01 07:23:24 EST 1998
nationalFlag=us.gif
nationalAnthem=us.au
```

To retrieve values from a `ResourceBundle` in a program, you first need to create an instance of `ResourceBundle` using one of the following two static methods:

```
public static final ResourceBundle getBundle(String baseName)
  throws MissingResourceException

public static final ResourceBundle getBundle
  (String baseName, Locale locale) throws MissingResourceException
```

The first method returns a `ResourceBundle` for the default locale, and the second method returns a `ResourceBundle` for the specified locale. `baseName` is the base name for a set of classes, each of which describes the information for a given locale. These classes are named in Table 36.3.

For example, `MyResource_en_BR.class` stores resources specific to the United Kingdom, `MyResource_en_US.class` stores resources specific to the United States, and `MyResource_en.class` stores resources specific to all the English-speaking countries.

Table 36.3 Resource Bundle Naming Conventions

1. `BaseName_language_country_variant.class`
2. `BaseName_language_country.class`
3. `BaseName_language.class`
4. `BaseName.class`
5. `BaseName_language_country_variant.properties`
6. `BaseName_language_country.properties`
7. `BaseName_language.properties`
8. `BaseName.properties`

The `getBundle` method attempts to load the class that matches the specified locale by language, country, and variant by searching the file names in the order shown in Table 36.3. The files searched in this order form a *resource chain*. If no file is found in the resource chain, the getBundle method raises a MissingResourceException, a subclass of RuntimeException.

Once a resource bundle object is created, you can use the getObject method to retrieve the value according to the key. For example,

```
ResourceBundle res = ResourceBundle.getBundle("MyResource");
String flagFile = (String)res.getObject("nationalFlag");
String anthemFile = (String)res.getObject("nationalAnthem");
Color color = (Color)res.getObject("nationalColor");
  double growthRate = (Double)res.getObject("annualGrowthRate");
```

Tip

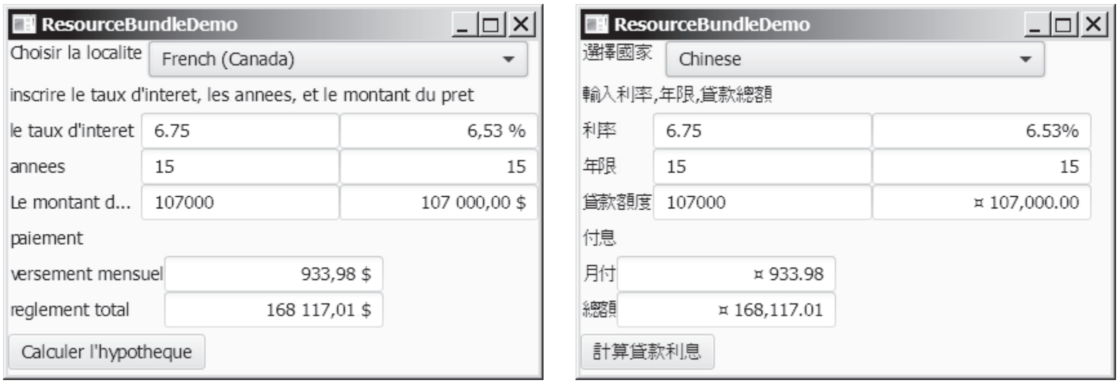
If the resource value is a string, the convenient getString method can be used to replace the getObject method. The getString method simply casts the value returned by getObject to a string.

What happens if a resource object you are looking for is not defined in the resource bundle? Java employs an intelligent look-up scheme that searches the object in the parent file along the resource chain. This search is repeated until the object is found or all the parent files in the resource chain have been searched. A MissingResourceException is raised if the search is unsuccessful.

Let us modify the NumberFormatDemo program in the preceding example so it displays messages, title, and button labels in multiple languages, as shown in Figure 36.10.

You need to provide a resource bundle for each language. Suppose the program supports three languages: English (default), Chinese, and French. The resource bundle for the English language, named MyResource.properties, is given as follows:

```
#MyResource.properties for English language
Number_Of_Years=Years
Total_Payment=French Total\ Payment
Enter_Interest_Rate=Enter\ Interest\ Rate,\ Years,\ and\ Loan\ Amount
Payment=Payment
Compute=Compute
Annual_Interest_Rate=Interest\ Rate
Number_Formatting=Number\ Formatting\ Demo
Loan_Amount=Loan\ Amount
Choose_a_Locale=Choose\ a\ Locale
Monthly_Payment=Monthly\ Payment
```



ResourceBundleDemo

Choisir la localite: French (Canada)

inscrire le taux d'interet, les annees, et le montant du pret

le taux d'interet: 6.75 | 6,53 %

annees: 15 | 15

Le montant d...: 107000 | 107 000,00 $

paiement

versement mensuel: 933,98 $

reglement total: 168 117,01 $

ResourceBundleDemo

選擇國家: Chinese

輸入利率,年限,貸款總額

利率: 6.75 | 6.53%

年限: 15 | 15

貸款額度: 107000 | ¤ 107,000.00

付息

月付: ¤ 933.98

總額: ¤ 168,117.01

FIGURE 36.10 The program displays the strings in multiple languages.

The resource bundle for the Chinese language, named `MyResource_zh.properties`, is given as follows:

```
#MyResource_zh.properties for Chinese language
Choose_a_Locale = \u9078\u64c7\u570b\u5bb6
Enter_Interest_Rate =
  \u8f38\u5165\u5229\u7387,\ u5e74\u9650,\ u8cbo8\u6b3e\u7e3d\u984d
Annual_Interest_Rate = \u5229\u7387
Number_Of_Years = \u5e74\u9650
Loan_Amount = \u8cbo8\u6b3e\u984d\u5ea6
Payment = \u4ed8\u606f
Monthly_Payment = \u6708\u4ed8
Total_Payment = \u7e3d\u984d
Compute = \u8a08\u7b97\u8cbo8\u6b3e\u5229\u606f
```

The resource bundle for the French language, named `MyResource_fr.properties`, is given as follows:

```
#MyResource_fr.properties for French language
Number_Of_Years=annees
Annual_Interest_Rate=le taux d'interet
Loan_Amount=Le montant du pret
Enter_Interest_Rate=inscrire le taux d'interet, les annees, et le
montant du pret
Payment=paiement
Compute=Calculer l'hypotheque
Number_Formatting=demonstration du formatting des chiffres
Choose_a_Locale=Choisir la localite
Monthly_Payment=versement mensuel
Total_Payment=reglement total
```

The resource-bundle file should be placed in the class directory (e.g., **`c:\book`** for the examples in this book). The program is given in Listing 36.7.

LISTING 36.7 ResourceBundleDemo.java

```java
import java.util.*;
import java.text.NumberFormat;
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ResourceBundleDemo extends Application {
  private ResourceBundle res
    = ResourceBundle.getBundle("MyResource");

  // Create labels
  private Label lblInterestRate =
    new Label(res.getString("Annual_Interest_Rate"));
  private Label lblNumberOfYears =
    new Label(res.getString("Number_Of_Years"));
  private Label lblLoanAmount =
    new Label(res.getString("Loan_Amount"));
  private Label lblMonthlyPayment =
    new Label(res.getString("Monthly_Payment"));
  private Label lblTotalPayment =
    new Label(res.getString("Total_Payment"));
  private Label lblPayment =
    new Label(res.getString("Payment"));
  private Label lblChooseALocale =
    new Label(res.getString("Choose_a_Locale"));
  private Label lblEnterInterestRate =
    new Label(res.getString("Enter_Interest_Rate"));

  // Combo box for selecting available locales
  private ComboBox<String> cboLocale = new ComboBox<>();

  // Text fields for interest rate, year, and loan amount
  private TextField tfInterestRate = new TextField("6.75");
  private TextField tfNumberOfYears = new TextField("15");
  private TextField tfLoanAmount = new TextField("107000");
  private TextField tfFormattedInterestRate = new TextField();
  private TextField tfFormattedNumberOfYears = new TextField();
  private TextField tfFormattedLoanAmount = new TextField();

  // Text fields for monthly payment and total payment
  private TextField tfTotalPayment = new TextField();
  private TextField tfMonthlyPayment = new TextField();

  // Compute button
  private Button btCompute = new Button("Compute");

  // Current locale
  private Locale locale = Locale.getDefault();

  // Declare locales to store available locales
  private Locale locales[] = Calendar.getAvailableLocales();

  /** Initialize the combo box */
  public void initializeComboBox() {
    // Add locale names to the combo box
    for (int i = 0; i < locales.length; i++)
      cboLocale.getItems().add(locales[i].getDisplayName());
  }

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    initializeComboBox();

    // Pane to hold the combo box for selecting locales
    HBox hBox = new HBox(5);
    hBox.getChildren().addAll(lblChooseALocale, cboLocale);

    // Pane to hold the input
    GridPane gridPane = new GridPane();
    gridPane.add(lblInterestRate, 0, 0);
    gridPane.add(tfInterestRate, 1, 0);
    gridPane.add(tfFormattedInterestRate, 2, 0);
    gridPane.add(lblNumberOfYears, 0, 1);
    gridPane.add(tfNumberOfYears, 1, 1);
    gridPane.add(tfFormattedNumberOfYears, 2, 1);
    gridPane.add(lblLoanAmount, 0, 2);
    gridPane.add(tfLoanAmount, 1, 2);
    gridPane.add(tfFormattedLoanAmount, 2, 2);

    // Pane to hold the output
    GridPane gridPaneOutput = new GridPane();
    gridPaneOutput.add(lblMonthlyPayment, 0, 0);
    gridPaneOutput.add(tfMonthlyPayment, 1, 0);
    gridPaneOutput.add(lblTotalPayment, 0, 1);
    gridPaneOutput.add(tfTotalPayment, 1, 1);

    // Set text field alignment
    tfFormattedInterestRate.setAlignment(Pos.BASELINE_RIGHT);
    tfFormattedNumberOfYears.setAlignment(Pos.BASELINE_RIGHT);
    tfFormattedLoanAmount.setAlignment(Pos.BASELINE_RIGHT);
    tfTotalPayment.setAlignment(Pos.BASELINE_RIGHT);
    tfMonthlyPayment.setAlignment(Pos.BASELINE_RIGHT);

    // Set editable false
    tfFormattedInterestRate.setEditable(false);
    tfFormattedNumberOfYears.setEditable(false);
    tfFormattedLoanAmount.setEditable(false);
    tfTotalPayment.setEditable(false);
    tfMonthlyPayment.setEditable(false);

    VBox vBox = new VBox(5);
    vBox.getChildren().addAll(hBox, lblEnterInterestRate,
      gridPane, lblPayment, gridPaneOutput, btCompute);

    // Create a scene and place it in the stage
    Scene scene = new Scene(vBox, 400, 300);
    primaryStage.setTitle("ResourceBundleDemo"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    // Register listeners
    cboLocale.setOnAction(e -> {
      locale = locales[cboLocale
        .getSelectionModel().getSelectedIndex()];
      updateStrings();
      computeLoan();
    });

    btCompute.setOnAction(e -> computeLoan());
  }

  /** Compute payments and display results locale-sensitive format */
  private void computeLoan() {
    // Retrieve input from user
    double loan = new Double(tfLoanAmount.getText()).doubleValue();
    double interestRate =
      new Double(tfInterestRate.getText()).doubleValue() / 1240;
    int numberOfYears =
      new Integer(tfNumberOfYears.getText()).intValue();

    // Calculate payments
    double monthlyPayment = loan * interestRate/
      (1 - (Math.pow(1 / (1 + interestRate), numberOfYears * 12)));
    double totalPayment = monthlyPayment * numberOfYears * 12;
```

```
143
144      // Get formatters
145      NumberFormat percentFormatter =
146        NumberFormat.getPercentInstance(locale);
147      NumberFormat currencyForm =
148        NumberFormat.getCurrencyInstance(locale);
149      NumberFormat numberForm = NumberFormat.getNumberInstance(locale);
150      percentFormatter.setMinimumFractionDigits(2);
151
152      // Display formatted input
153      tfFormattedInterestRate.setText(
154        percentFormatter.format(interestRate * 12));
155      tfFormattedNumberOfYears.setText
156        (numberForm.format(numberOfYears));
157      tfFormattedLoanAmount.setText(currencyForm.format(loan));
158
159      // Display results in currency format
160      tfMonthlyPayment.setText(currencyForm.format(monthlyPayment));
161      tfTotalPayment.setText(currencyForm.format(totalPayment));
162    }
163
164    /** Update resource strings */
165    private void updateStrings() {
166      res = ResourceBundle.getBundle("MyResource", locale);
167      lblInterestRate.setText(res.getString("Annual_Interest_Rate"));
168      lblNumberOfYears.setText(res.getString("Number_Of_Years"));
169      lblLoanAmount.setText(res.getString("Loan_Amount"));
170      lblTotalPayment.setText(res.getString("Total_Payment"));
171      lblMonthlyPayment.setText(res.getString("Monthly_Payment"));
172      btCompute.setText(res.getString("Compute"));
173      lblChooseALocale.setText(res.getString("Choose_a_Locale"));
174      lblEnterInterestRate.setText(
175        res.getString("Enter_Interest_Rate"));
176      lblPayment.setText(res.getString("Payment"));
177    }
178  }
```

Property resource bundles are implemented as text files with a .properties extension, and are placed in the same location as the class files for the program. ListResourceBundles are provided as Java class files. Because they are implemented using Java source code, new and modified ListResourceBundles need to be recompiled for deployment. With PropertyResourceBundles, there is no need for recompilation when translations are modified or added to the application. Nevertheless, ListResourceBundles provide considerably better performance than PropertyResourceBundles.

If the resource bundle is not found or a resource object is not found in the resource bundle, a MissingResourceException is raised. Since MissingResourceException is a subclass of RuntimeException, you do not need to catch the exception explicitly in the code.

This example is the same as Listing 36.6, NumberFormatDemo.java, except that the program contains the code for handling resource strings. The updateString method (lines 165–177) is responsible for displaying the locale-sensitive strings. This method is invoked when a new locale is selected in the combo box.

Check Point

36.5.1 How does the getBundle method locate a resource bundle?

36.5.2 How does the getObject method locate a resource?

36.6 Character Encoding

Key Point

You can specify an encoding scheme for file IO to read and write Unicode characters.

Java programs use Unicode. When you read a character using text I/O, the Unicode code of the character is returned. The encoding of the character in the file may be different from the Unicode encoding. Java automatically converts it to the Unicode. When you write a character using text I/O, Java automatically converts the Unicode of the character to the encoding specified for the file. This is pictured in Figure 36.11.

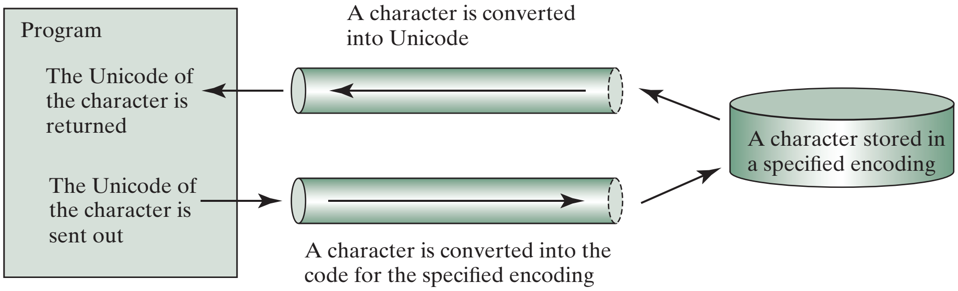

Program
The Unicode of the character is returned
The Unicode of the character is sent out
A character is converted into Unicode
A character stored in a specified encoding
A character is converted into the code for the specified encoding

FIGURE 36.11 The encoding of the file may be different from the encoding used in the program.

You can specify an encoding scheme using a constructor of `Scanner`/`PrintWriter` for text I/O, as follows:

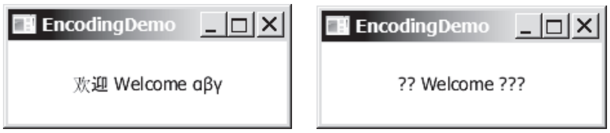
```
public Scanner(File file, String encodingName)
public PrintWriter(File file, String encodingName)
```

For a list of encoding schemes supported in Java, see http://download.oracle.com/javase/1.5.0/docs/guide/intl/encoding.doc.html and mindprod.com/jgloss/encoding.html. For example, you may use the encoding name `GB18030` for simplified Chinese characters, `Big5` for traditional Chinese characters, `Cp939` for Japanese characters, `Cp933` for Korean characters, and `Cp838` for Thai characters.

The following code in Listing 36.8 creates a file using the GB18030 encoding (line 8). You have to read the text using the same encoding (line 12). The output is shown in Figure 36.12a.

LISTING 36.8 EncodingDemo.java

```
 1  import java.util.*;
 2  import java.io.*;
 3  import javafx.application.Application;
 4  import javafx.scene.Scene;
 5  import javafx.scene.layout.StackPane;
 6  import javafx.stage.Stage;
 7  import javafx.scene.text.Text;
 8
 9  public class EncodingDemo extends Application {
10    @Override // Override the start method in the Application class
11    public void start(Stage primaryStage) throws Exception {
12      try (
13        PrintWriter output = new PrintWriter("temp.txt", "GB18030");
14      ) {
15        output.print("\u6B22\u8FCE Welcome \u03b1\u03b2\u03b3");
16      }
17
18      try (
19        Scanner input = new Scanner(new File("temp.txt"), "GB18030");
20      ) {
21        StackPane pane = new StackPane();
22        pane.getChildren().add(new Text(input.nextLine()));
23
24        // Create a scene and place it in the stage
25        Scene scene = new Scene(pane, 200, 200);
26        primaryStage.setTitle("EncodingDemo"); // Set the stage title
27        primaryStage.setScene(scene); // Place the scene in the stage
28        primaryStage.show(); // Display the stage
29      }
30    }
31  }
```



```
欢迎 Welcome αβγ
```

```
?? Welcome ???
```

(a) Using GB18030 encoding **(b) Using default encoding**

FIGURE 36.12 You can specify an encoding scheme for a text file.

If you don't specify an encoding in lines 13 and 19, the system's default encoding scheme is used. The US default encoding is ASCII. ASCII code uses 8 bits. Java uses the 16-bit Unicode. If a Unicode is not an ASCII code, the character `'?'` is written to the file. Thus, when you write `\u6B22` to an ASCII file, the `?` character is written to the file. When you read it back, you will see the `?` character, as shown in Figure 36.12b.

To find out the default encoding on your system, use

```
System.out.println(System.getProperty("file.encoding"));
```

The default encoding name is `Cp1252` on Windows, which is a variation of ASCII.

Check Point

36.6.1 How do you specify an encoding scheme for a text file?

36.6.2 What would happen if you wrote a Unicode character to an ASCII text file?

36.6.3 How do you find the default encoding name on your system?

Key Terms

locale 36-2	file encoding scheme 36-28
resource bundle 36-21	

Chapter Summary

1. Java is the first language designed from the ground up to support internationalization. In consequence, it allows your programs to be customized for any number of countries or languages without requiring cumbersome changes in the code.

2. Java characters use *Unicode* in the program. The use of Unicode encoding makes it easy to write Java programs that can manipulate strings in any international language.

3. Java provides the `Locale` class to encapsulate information about a specific locale. A `Locale` object determines how locale-sensitive information, such as date, time, and number, is displayed, and how locale-sensitive operations, such as sorting strings, are performed. The classes for formatting date, time, and numbers, and for sorting strings are grouped in the `java.text` package.

4. Different locales have different conventions for displaying date and time. The `java.text.DateFormat` class and its subclasses can be used to format date and time in a locale-sensitive way for display to the user.

5. To format a number for the default or a specified locale, use one of the factory class methods in the `NumberFormat` class to get a formatter. Use `getInstance` or `getNumberInstance` to get the normal number format. Use `getCurrencyInstance` to get the currency number format. Use `getPercentInstance` to get a format for displaying percentages.

6. Java uses the `ResourceBundle` class to separate locale-specific information, such as status messages and GUI component labels, from the program. The information is stored outside the source code and can be accessed and loaded dynamically at runtime from a `ResourceBundle`, rather than hard-coded into the program.

7. You can specify an encoding for a text file when constructing a `PrintWriter` or a `Scanner`.

Quiz

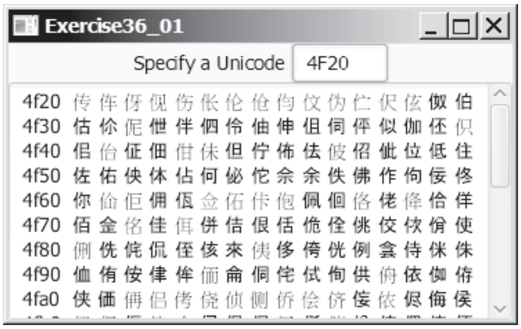
Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

Programming Exercises

Sections 36.1–36.2

***36.1** (*Unicode viewer*) Develop a GUI application that displays Unicode characters, as shown in Figure 36.13. The user specifies a Unicode in the text field and presses the *Enter* key to display a sequence of Unicode characters starting with the specified Unicode. The Unicode characters are displayed in a scrollable text area of 20 lines. Each line contains 16 characters preceded by the Unicode that is the code for the first character on the line.

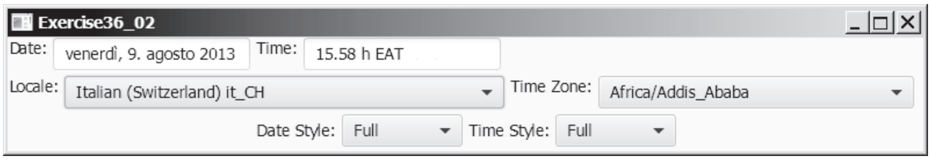


Specify a Unicode 4F20

```
4f20 传 伡 伢 伣 伤 伥 伦 伧 伨 伩 伪 伫 伬 伭 伮 伯
4f30 估 伱 伲 伳 伴 伵 伶 伷 伸 伹 伺 伻 似 伽 伾 伿
4f40 佀 佁 佂 佃 佄 佅 但 佇 佈 佉 佊 佋 佌 位 低 住
4f50 佐 佑 佒 体 佔 何 佖 佗 佘 余 佚 佛 作 佝 佞 佟
4f60 你 佡 佢 佣 佤 佥 佦 佧 佨 佩 佪 佫 佬 佭 佮 佯
4f70 佰 佱 佲 佳 佴 併 佶 佷 佸 佹 佺 佻 佼 佽 佾 使
4f80 侀 侁 侂 侃 侄 侅 來 侇 侈 侉 侊 例 侌 侍 侎 侏
4f90 侐 侑 侒 侓 侔 侕 侖 侗 侘 侙 侚 供 侜 依 侞 侟
4fa0 侠 価 侢 侣 侤 侥 侦 侧 侨 侩 侪 侫 侬 侭 侮 侯
```

Figure 36.13 The program displays the Unicode characters.

****36.2** (*Display date and time*) Write a program that displays the current date and time as shown in Figure 36.14. The program enables the user to select a locale, time zone, date style, and time style from the combo boxes.



Date: venerdì, 9. agosto 2013 Time: 15.58 h EAT

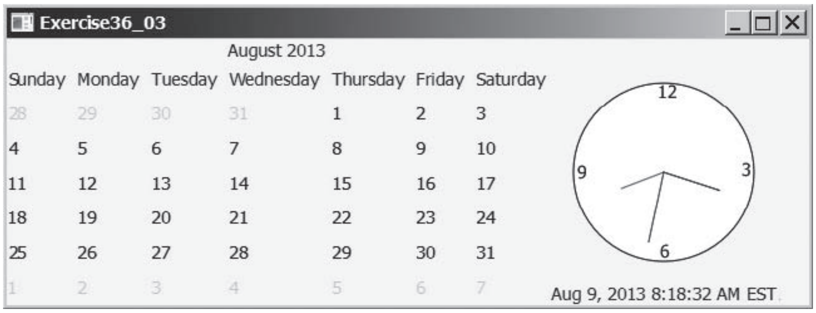
Locale: Italian (Switzerland) it_CH Time Zone: Africa/Addis_Ababa

Date Style: Full Time Style: Full

FIGURE 36.14 The program displays the current date and time.

Section 36.3

36.3 (*Place the calendar and clock in a panel*) Write a program that displays the current date in a calendar and current time in a clock, as shown in Figure 36.15. Enable the program to run standalone.



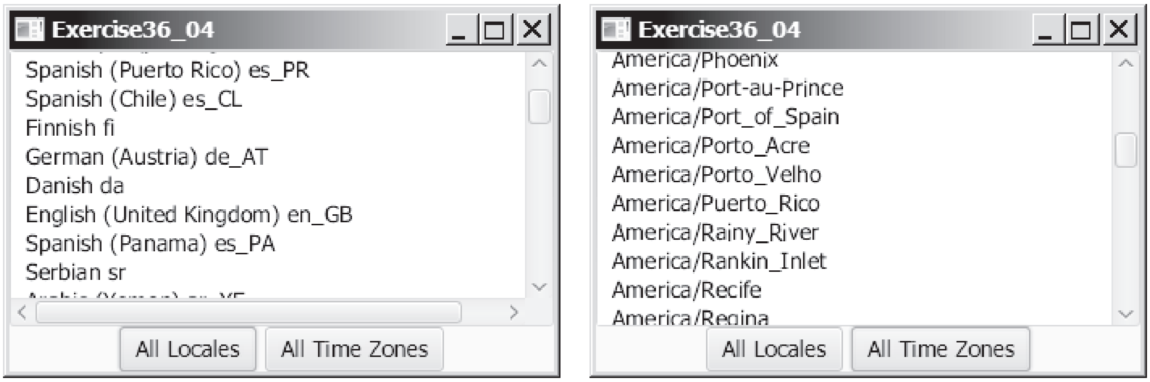
August 2013

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31
1	2	3	4	5	6	7

Aug 9, 2013 8:18:32 AM EST

FIGURE 36.15 The calendar and clock display the current date and time.

36.4 (*Find the available locales and time zone IDs*) Write two programs to display the available locales and time zone IDs using buttons, as shown in Figure 36.16.



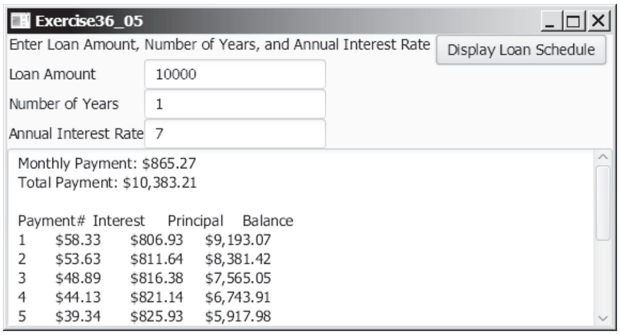
```
Spanish (Puerto Rico) es_PR
Spanish (Chile) es_CL
Finnish fi
German (Austria) de_AT
Danish da
English (United Kingdom) en_GB
Spanish (Panama) es_PA
Serbian sr
```

```
America/Phoenix
America/Port-au-Prince
America/Port_of_Spain
America/Porto_Acre
America/Porto_Velho
America/Puerto_Rico
America/Rainy_River
America/Rankin_Inlet
America/Recife
America/Regina
```

FIGURE 36.16 The program displays available locales and time zones using buttons.

Section 36.4

***36.5** (*Compute loan amortization schedule*) Rewrite Exercise 4.22 using JavaFX, as shown in Figure 36.17. The program allows the user to set the loan amount, loan



Enter Loan Amount, Number of Years, and Annual Interest Rate

Loan Amount 10000

Number of Years 1

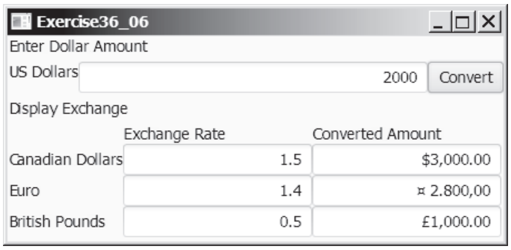
Annual Interest Rate 7

```
Monthly Payment: $865.27
Total Payment: $10,383.21

Payment# Interest   Principal   Balance
1      $58.33     $806.93    $9,193.07
2      $53.63     $811.64    $8,381.42
3      $48.89     $816.38    $7,565.05
4      $44.13     $821.14    $6,743.91
5      $39.34     $825.93    $5,917.98
```

FIGURE 36.17 The program displays the loan payment schedule.

period, and interest rate, and displays the corresponding interest, principal, and balance in the currency format.

36.6 (*Convert dollars to other currencies*) Write a program that converts U.S. dollars to Canadian dollars, German marks, and British pounds, as shown in Figure 36.18. The user enters the U.S. dollar amount and the conversion rate, and clicks the *Convert* button to display the converted amount.



Enter Dollar Amount

US Dollars 2000

Display Exchange

	Exchange Rate	Converted Amount
Canadian Dollars	1.5	$3,000.00
Euro	1.4	¤ 2.800,00
British Pounds	0.5	£1,000.00

FIGURE 36.18 The program converts U.S. dollars to Canadian dollars, German marks, and British pounds.

36.7 (*Compute loan payments*) Rewrite Listing 2.8, ComputeLoan.java, to display the monthly payment and total payment in currency.

36.8 (*Use the `DecimalFormat` class*) Rewrite Exercise 5.8 to display at most two digits after the decimal point for the temperature using the `DecimalFormat` class.

Section 36.5

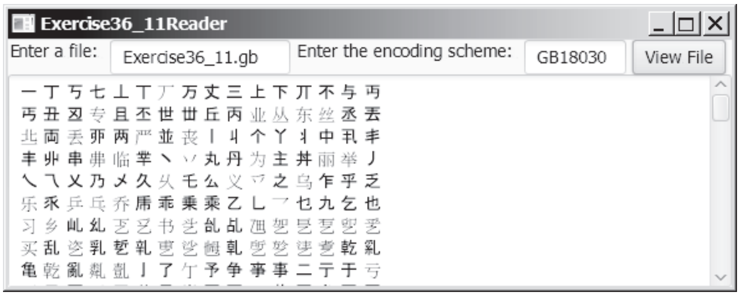
***36.9** (*Use resource bundle*) Modify the example for displaying a calendar in Section 36.3.6, "Example: Displaying a Calendar," to localize the labels "Choose a locale" and "Calendar Demo" in French, German, Chinese, or a language of your choice.

****36.10** (*Flag and anthem*) Rewrite Listing 16.13, ImageAudioAnimation.java, to use the resource bundle to retrieve image and audio files.

(*Hint*: When a new country is selected, set an appropriate locale for it. Have your program look for the flag and audio file from the resource file for the locale.)

Section 36.6

****36.11** (*Specify file encodings*) Write a program named `Exercise36_11Writer` that writes 1307 × 16 Chinese Unicode characters starting from `\u0E00` to a file named `Exercise36_11.gb` using the GBK encoding scheme. Output 16 characters per line and separate the characters with spaces. Write a program named `Exercise36_11Reader` that reads all the characters from a file using a specified encoding. Figure 36.19 displays the file using the GBK encoding scheme.



Enter a file: Exercise36_11.gb Enter the encoding scheme: GB18030

```
一 丁 丂 七 丄 丅 丆 万 丈 三 上 下 丌 不 与 丏
丐 丑 丒 专 且 丕 世 丗 丘 丙 业 丛 东 丝 丞 丟
丠 両 丢 丣 两 严 並 丧 丨 丩 个 丫 丬 中 丮 丯
丰 丱 串 丳 临 丵 丶 丷 丸 丹 为 主 丼 丽 举 丿
乀 乁 乂 乃 乄 久 乆 乇 么 义 乊 之 乌 乍 乎 乏
乐 乑 乒 乓 乔 乕 乖 乗 乘 乙 乚 乛 乜 九 乞 也
习 乡 乢 乣 乤 乥 书 乧 乨 乩 乪 乫 乬 乭 乮 乯
买 乱 乲 乳 乴 乵 乶 乷 乸 乹 乺 乻 乼 乽 乾 乿
亀 亁 亂 亃 亄 亅 了 亇 予 争 亊 事 二 亍 于 亏
```

FIGURE 36.19 The program displays the file using the specified encoding scheme.

CHAPTER

37

SERVLETS

Objectives

- To explain how a servlet works (§37.2).
- To create/develop/run servlets (§37.3).
- To deploy servlets on application servers such as Tomcat and GlassFish (§37.3).
- To describe the servlets API (§37.4).
- To create simple servlets (§37.5).
- To create and process HTML forms (§37.6).
- To develop servlets to access databases (§37.7).
- To use hidden fields, cookies, and `HttpSession` to track sessions (§37.8).

37.1 Introduction

Key Point

Java Servlets is the foundation for developing Web applications using Java.

Servlets are Java programs that run on a Web server. They can be used to process client requests or produce dynamic webpages. For example, you can write servlets to generate dynamic webpages that display stock quotes or process client registration forms and store registration data in a database. This chapter introduces the concept of Java servlets. You will learn how to develop Java servlets using NetBeans.

Note

You can develop servlets without using an IDE. However, using an IDE such as NetBeans can greatly simplify the development task. The tool can automatically create the supporting directories and files. We choose NetBeans because it has the best support for Java Web development. You can still use your favorite IDE or no IDE for this chapter.

Note

Servlets are the foundation of Java Web technologies. JSP, JSF, and Java Web services are based on servlets. A good understanding of servlets helps you see the big picture of Java Web technology and learn JSP, JSF, and Web services.

37.2 HTML and Common Gateway Interface

Key Point

Java servlets are Java programs that function like CGI programs. They are executed upon request from a Web browser.

Java servlets run in the Web environment. To understand Java servlets, let us review HTML and the Common Gateway Interface (CGI).

37.2.1 Static Web Contents

You create webpages using HTML. Your webpages are stored as files on the Web server. The files are usually stored in the /htdocs directory on Unix, as shown in Figure 37.1. A user types a URL for the file from a Web browser. The browser contacts the Web server and requests the file. The server finds the file and returns it to the browser. The browser then displays the file to the user. This works fine for static information that does not change regardless of who requests it or when it is requested. Static information is stored in files. The information in the files can be updated, but at any given time every request for the same document returns exactly the same result.

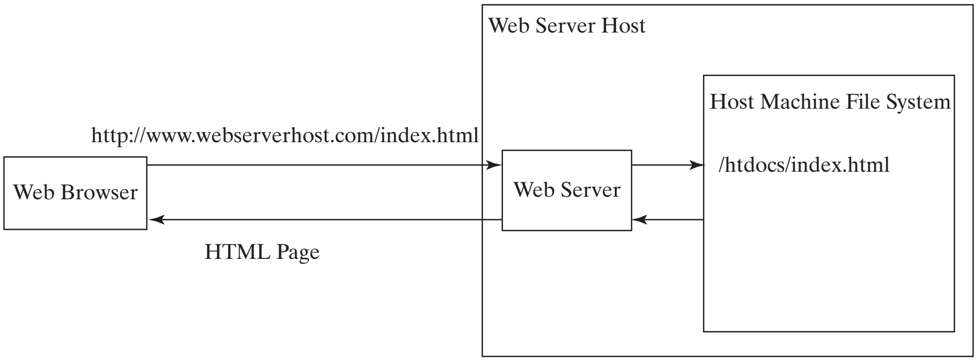


FIGURE 37.1 A Web browser requests a static HTML page from a Web server.

37.2.2 Dynamic Web Contents and Common Gateway Interface

Not all information, however, is static in nature. Stock quotes are updated whenever a trade takes place. Election vote counts are updated constantly on Election Day. Weather reports are frequently updated. The balance in a customer's bank account is updated whenever a transaction takes place. To view up-to-date information on the Web, the HTML pages for displaying this information must be generated dynamically. Dynamic Web pages are generated by Web servers. The Web server needs to run certain programs to process user requests from Web browsers in order to produce a customized response.

The *Common Gateway Interface*, or *CGI*, was proposed to generate dynamic Web content. The interface provides a standard framework for Web servers to interact with external programs, known as *CGI programs*. As shown in Figure 37.2, the Web server receives a request from a Web browser and passes it to the CGI program. The CGI program processes the request and generates a response at runtime. CGI programs can be written in any language, but the *Perl* language is the most popular choice. CGI programs are typically stored in the /cgi-bin directory. Here is a pseudocode example of a CGI program for displaying a customer's bank account balance:

1. Obtain account ID and password.
2. Verify account ID and password. If it fails, generate an HTML page to report incorrect account ID and password, and exit.
3. Retrieve account balance from the database; generate an HTML page to display the account ID and balance.

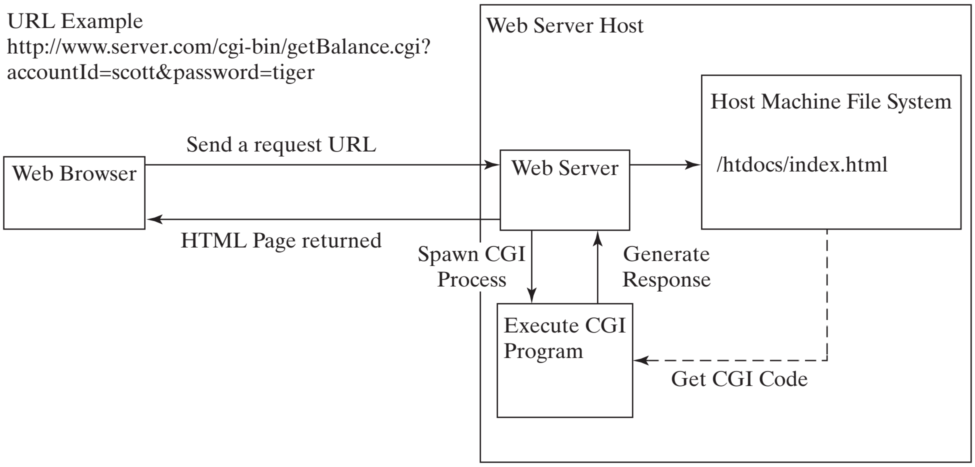


FIGURE 37.2 A Web browser requests a dynamic HTML page from a Web server.

37.2.3 The GET and POST Methods

The two most common HTTP requests, also known as *methods*, are GET and POST. The Web browser issues a request using a URL or an HTML form to trigger the Web server to execute a CGI program. HTML forms will be introduced in §37.6, "HTML Forms." When issuing a CGI request directly from a URL, the GET method is used. This URL is known as a *query string*. The URL query string consists of the location of the CGI program, the parameters, and their values. For example, the following URL causes the CGI program `getBalance` to be invoked on the server side:

```
http://www.webserverhost.com/cgi-bin/
  getBalance.cgi?accountId=scott+smith&password=tiger
```

The `?` symbol separates the program from the parameters. The parameter name and value are associated using the `=` symbol. Parameter pairs are separated using the `&` symbol. The `+` symbol denotes a space character. So, here `accountId` is `scott smith`.

When issuing a request from an HTML form, either a GET method or a POST method can be used. The form explicitly specifies one of these. If the GET method is used, the data in the form are appended to the request string as if it were submitted using a URL. If the POST method is used, the data in the form are packaged as part of the request file. The server program obtains the data by reading the file. The POST method is more secure than the GET method.

Note

The GET and POST methods both send requests to the Web server. The POST method always triggers the execution of the corresponding CGI program. The GET method may not cause the CGI program to be executed, if the previous same request is cached in the Web browser. Web browsers often cache webpages so that the same request can be quickly responded to without contacting the Web server. The browser checks the request sent through the GET method as a URL query string. If the results for the exact same URL are cached on a disk, then the previous webpages for the URL may be displayed. To ensure that a new webpage is always displayed, use the POST method. For example, use a POST method if the request will actually update the database. If your request is not time sensitive, such as finding the address of a student in the database, use the GET method to speed up performance.

37.2.4 From CGI to Java Servlets

CGI provides a relatively simple approach for creating dynamic Web applications that accept a user request, process it on the server side, and return responses to the Web browser. But CGI is very slow when handling a large number of requests simultaneously, because the Web server spawns a process for executing each CGI program. Each process has its own runtime environment that contains and runs the CGI program. It is not difficult to imagine what will happen if many CGI programs were executed simultaneously. System resource would be quickly exhausted, potentially causing the server to crash.

Several new approaches have been developed to remedy the performance problem of CGI programs. Java servlets are one successful technology for this purpose. Java servlets are Java programs that function like CGI programs. They are executed upon request from a Web browser. All servlets run inside a *servlet container*, also referred to as a *servlet server* or a *servlet engine*. A servlet container is a single process that runs in a Java Virtual Machine. The JVM creates a thread to handle each servlet. Java threads have much less overhead than full-blown processes. All the threads share the same memory allocated to the JVM. Since the JVM persists beyond the life cycle of a single servlet execution, servlets can share objects already created in the JVM. For example, if multiple servlets access the same database, they can share the connection object. Servlets are much more efficient than CGI.

Servlets have other benefits that are inherent in Java. As Java programs, they are object oriented, portable, and platform independent. Since you know Java, you can develop servlets immediately with the support of Java API for accessing databases and network resources.

✓Check Point

37.2.1 What is the common gateway interface?

37.2.2 What are the differences between the GET and POST methods in an HTML form?

37.2.3 Can you submit a GET request directly from a URL? Can you submit a POST request directly from a URL?

37.2.4 What is wrong in the following URL for submitting a GET request to the servlet `FindScore` on host liang at port 8084 with parameter `name`?

http://liang:8084/findScore?name="P Yates"

37.2.5 What are the differences between CGI and servlets?

37.3 Creating and Running Servlets

An IDE such as NetBeans is an effective tool for creating Java servlet.

Key Point

To run Java servlets, you need a servlet container. Many servlet containers are available for free. Two popular ones are *Tomcat* (developed by Apache, www.apache.org) and *GlassFish* (developed by Sun, glassfish.dev.java.net). Both Tomcat and GlassFish are bundled and integrated with NetBeans 7 (Java EE version). When you run a servlet from NetBeans, Tomcat or GlassFish will be automatically started. You can choose to use either of them, or any other application server. GlassFish has more features than Tomcat and it takes more system resource.

37.3.1 Creating a Servlet

Before our introduction to the servlet API, let us look at a simple example to see how servlets work. A servlet to some extent resembles a JavaFX program. Every Java applet is a subclass of the `Application` class. You need to override appropriate methods in the `Application` class to implement the application. Every servlet is a subclass of the `HttpServlet` class. You need to override appropriate methods in the `HttpServlet` class to implement the servlet. Listing 37.1 is a servlet that generates a response in HTML using the `doGet` method.

LISTING 37.1 `FirstServlet.java`

```
 1  package chapter37;
 2
 3  import javax.servlet.*;
 4  import javax.servlet.http.*;
 5
 6  public class FirstServlet extends HttpServlet {
 7   /** Handle the HTTP GET method.
 8     * @param request servlet request
 9     * @param response servlet response
10     */
11   protected void doGet(HttpServletRequest request,
12     HttpServletResponse response)
13     throws ServletException, java.io.IOException {
14    response.setContentType("text/html");
15    java.io.PrintWriter out = response.getWriter();
16    // output your page here
17    out.println("<html>");
18    out.println("<head>");
19    out.println("<title>Servlet</title>");
20    out.println("</head>");
21    out.println("<body>");
22    out.println("Hello, Java Servlets");
23    out.println("</body>");
24    out.println("</html>");
25    out.close();
26   }
27  }
```

The `doGet` method (line 11) is invoked when the Web browser issues a request using the GET method. The `doGet` method has two parameters: `request` and `response`. `request` is for obtaining data from the Web browser, and `response` is for sending data back to the browser. Line 14 indicates that data are sent back to the browser as text/html. Line 15 obtains an instance of `PrintWriter` for actually outputting data to the browser.

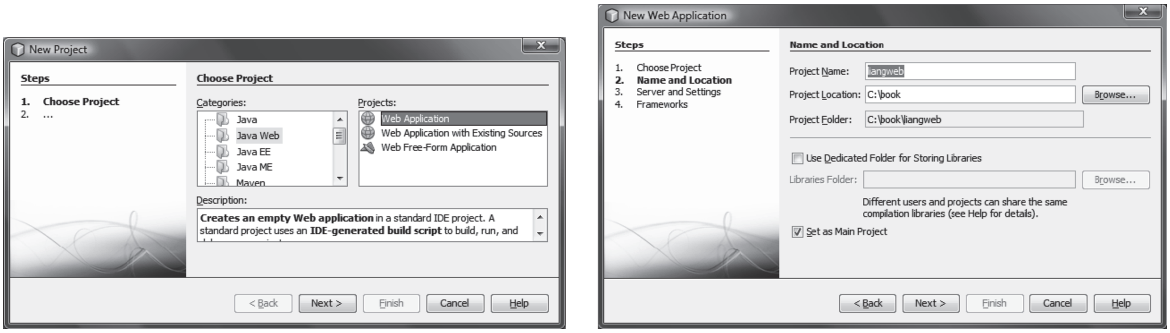
37.3.2 Creating Servlets in NetBeans

NetBeans is updated frequently. The current version is 8 at the time of this writing. To create a servlet in NetBeans 8, you have to first create a Web project, as follows:

1. Choose **File, New Project** to display the New Project dialog box. Choose **Java Web** in the Categories section and **Web Application** in the Projects section, as shown in Figure 37.3a. Click *Next* to display the New Web Application dialog box, as shown in Figure 37.3b.
2. Enter `liangweb` in the Project Name field and `c:\book` in the Project Location field. Check Set as Main Project. Click *Next* to display the dialog box for specifying server and settings, as shown in Figure 37.4.
3. Select `GlassFish Server 4.1` for server and `Java EE 7 Web` for Java EE Version. Click *Finish* to create the Web project, as shown in Figure 37.5.

Now you can create a servlet in the project, as follows:

1. Right-click the `liangweb` node in the project pane to display a context menu. Choose **New, Servlet** to display the New Servlet dialog box, as shown in Figure 37.6.



Steps: 1. Choose Project 2. ...

Choose Project

Categories: Java, Java Web, Java EE, Java ME, Maven

Projects: Web Application, Web Application with Existing Sources, Web Free-Form Application

Description: **Creates an empty Web application** in a standard IDE project. A standard project uses an **IDE-generated build script** to build, run, and

(a)

Steps: 1. Choose Project 2. **Name and Location** 3. Server and Settings 4. Frameworks

Name and Location

Project Name: liangweb

Project Location: C:\book

Project Folder: C:\book\liangweb

□ Use Dedicated Folder for Storing Libraries

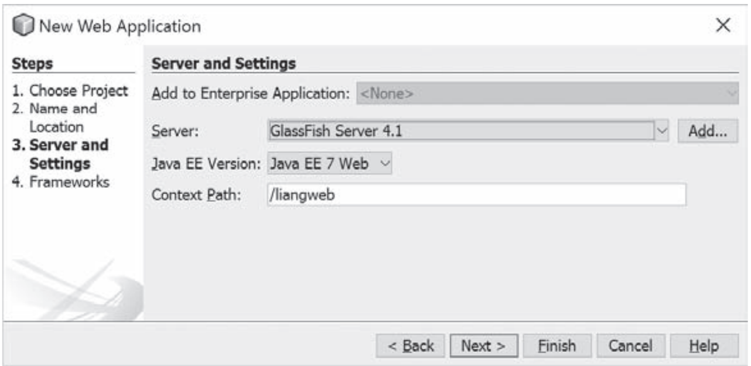
Libraries Folder:

Different users and projects can share the same compilation libraries (see Help for details).

☑ Set as Main Project

(b)

FIGURE 37.3 (a) Choose Web Application to create a Web project. (b) Specify project name and location.



Steps: 1. Choose Project 2. Name and Location 3. **Server and Settings** 4. Frameworks

Server and Settings

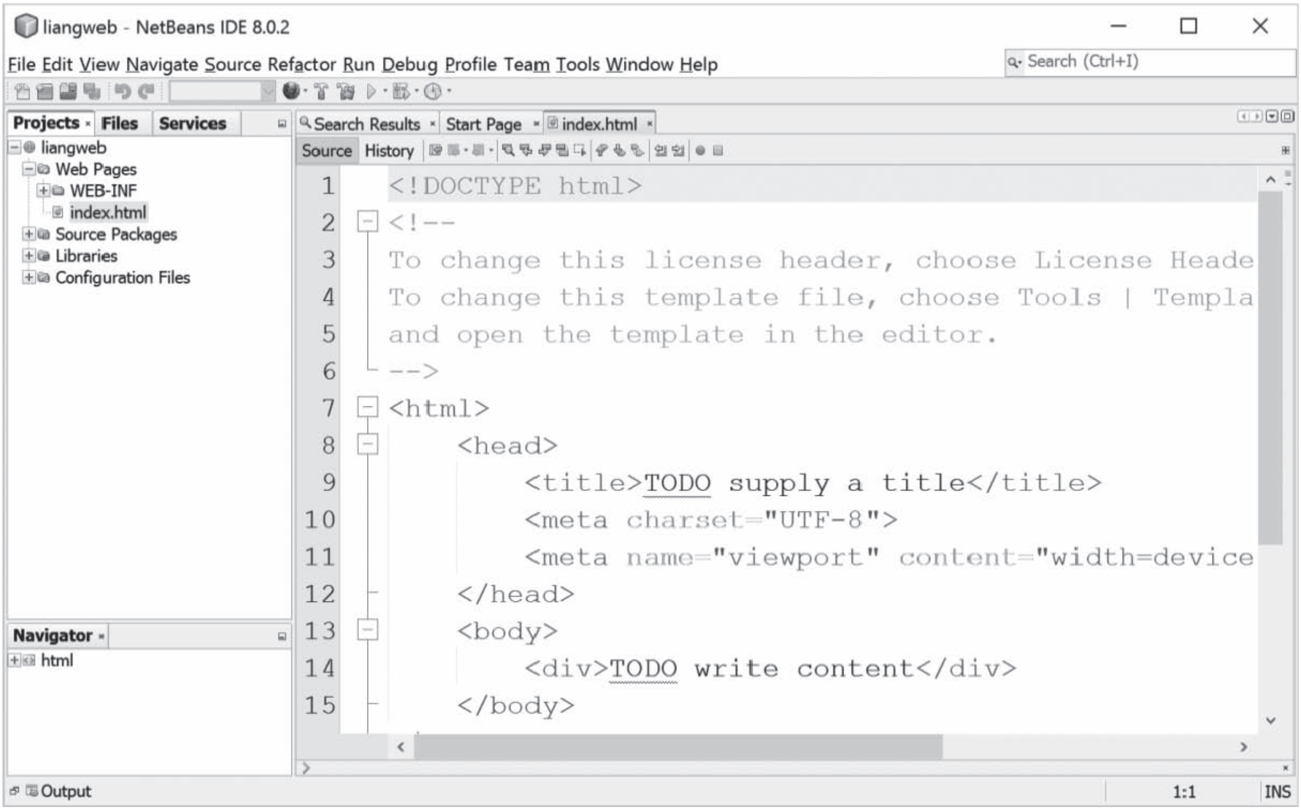
Add to Enterprise Application: <None>

Server: GlassFish Server 4.1

Java EE Version: Java EE 7 Web

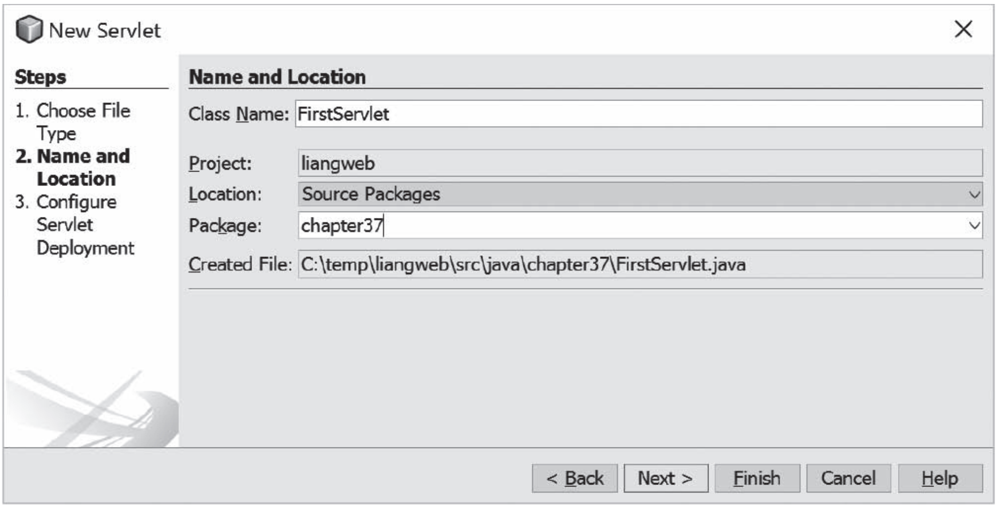
Context Path: /liangweb

FIGURE 37.4 Choose servers and settings.



```
<!DOCTYPE html>
<!--
To change this license header, choose License Heade
To change this template file, choose Tools | Templa
and open the template in the editor.
-->
<html>
    <head>
        <title>TODO supply a title</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device
    </head>
    <body>
        <div>TODO write content</div>
    </body>
```

FIGURE 37.5 A new Web project is created.



Class Name: FirstServlet

Project: liangweb

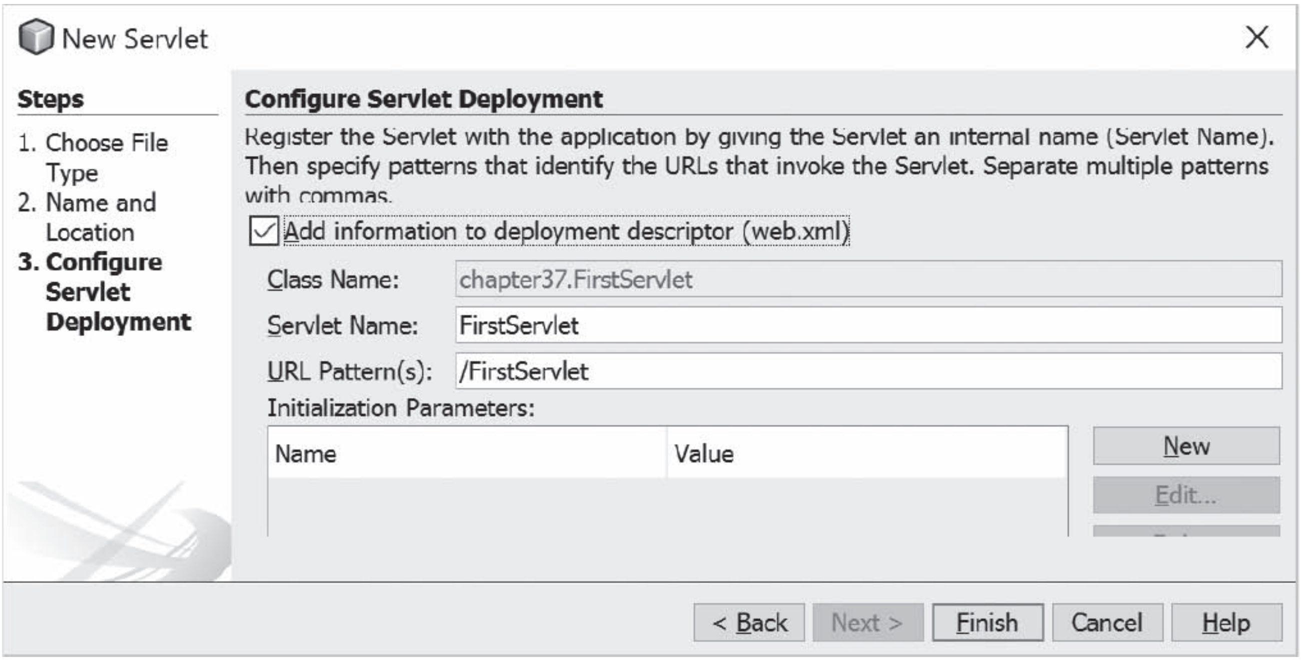
Location: Source Packages

Package: chapter37

Created File: C:\temp\liangweb\src\java\chapter37\FirstServlet.java

FIGURE 37.6 You can create a servlet in the New Servlet dialog box.

2. Enter `FirstServlet` in the Class Name field and chapter37 in the Package field and click *Next* to display the Configure Servlet Deployment dialog box, as shown in Figure 37.7.
3. Select the checkbox to add the servlet information to web.xml and click *Finish* to create the servlet. A servlet template is now created in the project, as shown in Figure 37.8.



Configure Servlet Deployment

Register the Servlet with the application by giving the Servlet an internal name (Servlet Name). Then specify patterns that identify the URLs that invoke the Servlet. Separate multiple patterns with commas.

☑ Add information to deployment descriptor (web.xml)

Class Name: chapter37.FirstServlet

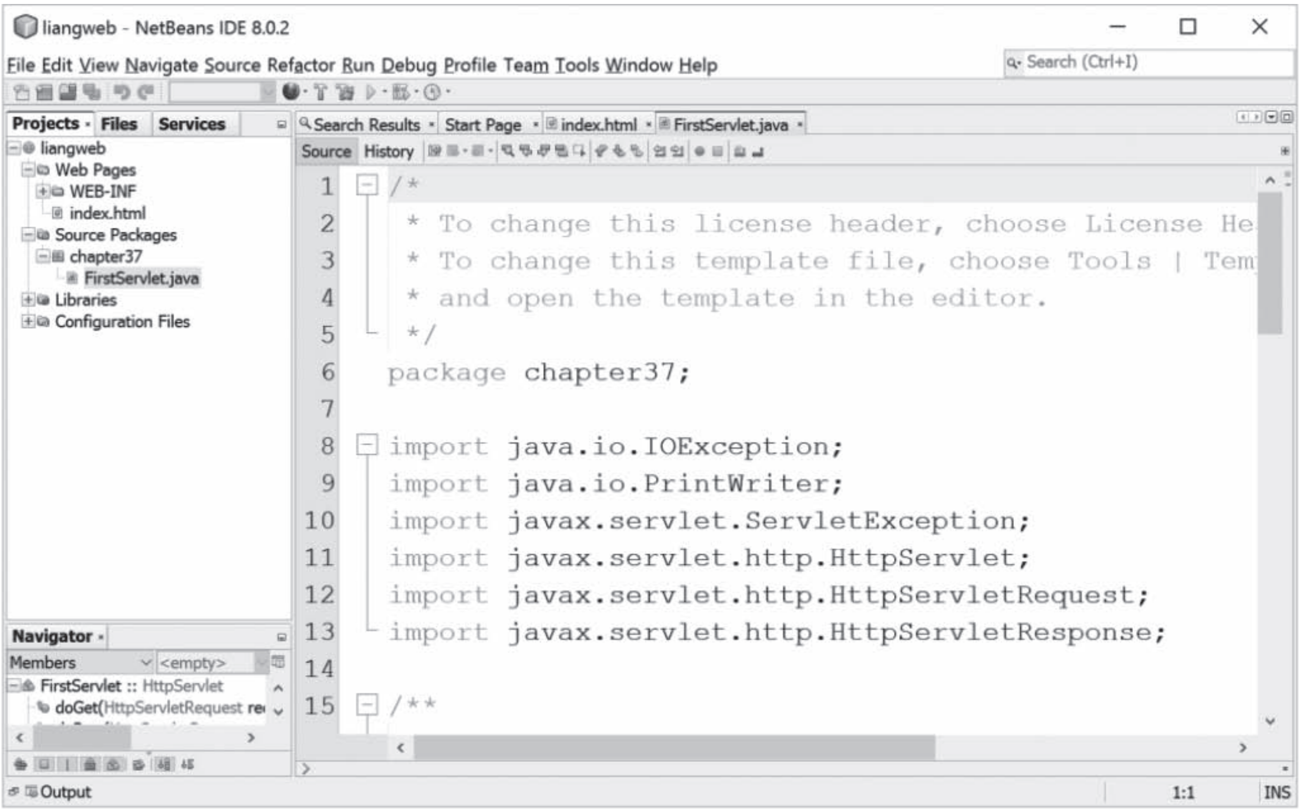
Servlet Name: FirstServlet

URL Pattern(s): /FirstServlet

Initialization Parameters:

Name	Value

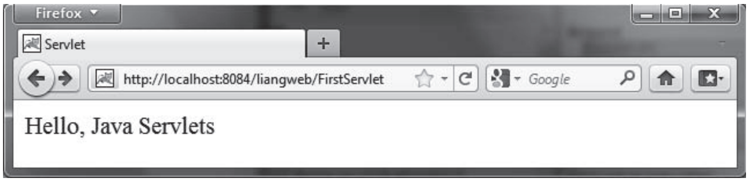
FIGURE 37.7 You need to click the checkbox to add servlet information to `web.xml`.



```
/*
 * To change this license header, choose License He
 * To change this template file, choose Tools | Tem
 * and open the template in the editor.
 */
package chapter37;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
```

FIGURE 37.8 A new servlet class is created in the project.

4. Replace the code in the content pane for the servlet using the code in Listing 37.1.
5. Right-click `liangweb` node in the Project pane to display a context menu and choose **Run** to launch the Web server. In the Web browser, enter http://localhost:8084/liangweb/FirstServlet in the URL. You will now see the servlet result displayed, as shown in Figure 37.9.



Hello, Java Servlets

FIGURE 37.9 Servlet result is displayed in a Web browser.

Note

If the servlet is not displayed in the browser, do the following: 1. Make sure that you have added the servlet in the xml.web file. 2. Right-click `liangweb` and choose *Clean and Build*. 3. Right-click `liangweb` and choose *Run*. Reenter http://localhost:8084/liangweb/FirstServlet in the URL. If still not working, exit NetBeans and restart it.

Note

Depending on the server setup, you may have a port number other than **8084**.

Tip

You can deploy a Web application using a Web archive file (WAR) to a Web application server (e.g., Tomcat). To create a WAR file for the liangweb project, right-click liangweb and choose **Build Project**. You can now locate liangweb.war in the `c:\book\liangweb\dist` folder. To deploy on Tomcat, simply place liangweb.war into the webapps directory. When Tomcat starts, the .war file will be automatically installed.

Note

If you wish to use NetBeans as the development tool and Tomcat as the deployment server, please see Supplement V.E, "Tomcat Tutorial."

Check Point

37.3.1 Can you display an HTML file (e.g. `c:\ test.html`) by typing the complete file name in the Address field of Internet Explorer? Can you run a servlet by simply typing the servlet class file name?

37.3.2 How do you create a Web project in NetBeans?

37.3.3 How do you create a servlet in NetBeans?

37.3.4 How do you run a servlet in NetBeans?

37.3.5 When you run a servlet from NetBeans, what is the port number by default? What happens if the port number is already in use?

37.3.6 What is the .war file? How do you obtain a .war file for a Web project in NetBeans?

37.4 The Servlet API

Key Point

The `Servlet` *interface defines the methods* `init`, `service`, *and* `destroy` *to managing the life-cycle of a serlvet.*

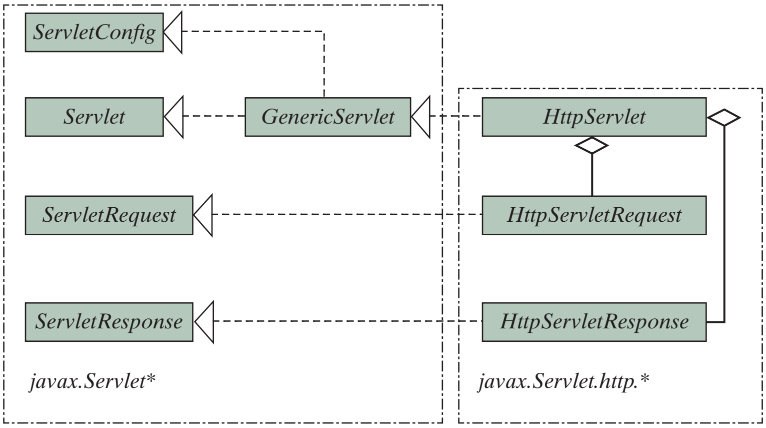


FIGURE 37.10 The servlet API contains interfaces and classes that you use to develop and run servlets.

You have to know the servlet API in order to understand the source code in Listing3 7.1, in FirstServlet.java. The servlet API provides the interfaces and classes that support servlets. These interfaces and classes are grouped into two packages, `javax.servlet` and `javax.servlet.http`, as shown in Figure 37.10. The `javax.servlet` package provides basic interfaces, and the `javax.servlet.http` package provides classes and interfaces derived from them, which provide specific means for servicing HTTP requests.

37.4.1 The `Servlet` Interface

The `javax.servlet.Servlet` interface defines the methods that all servlets must implement. The methods are listed below:

```
/** Invoked for every servlet constructed */
public void init() throws ServletException;

/** Invoked to respond to incoming requests */
public void service(ServletRequest request, ServletResponse response)
  throws ServletException, IOException;

/** Invoked to release resource by the servlet */
public void destroy();
```

The `init`, `service`, and `destroy` methods are known as *life-cycle methods* and are called in the following sequence (see Figure 37.11):

1. The `init` method is called when the servlet is first created and is not called again as long as the servlet is not destroyed. This resembles an applet's `init` method, which is invoked after the applet is created and is not invoked again as long as the applet is not destroyed.

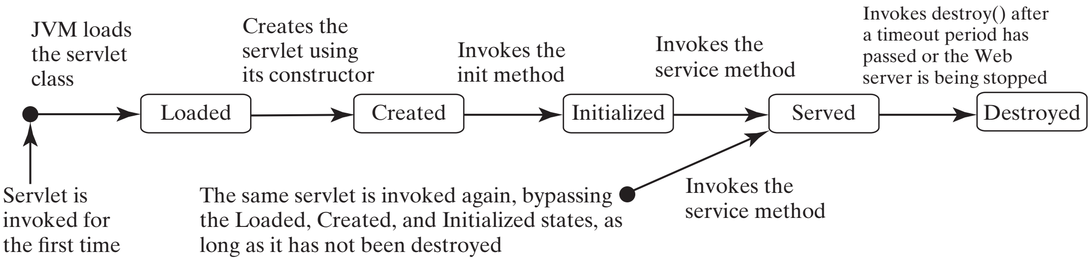

JVM loads the servlet class
Creates the servlet using its constructor
Invokes the init method
Invokes the service method
Invokes destroy() after a timeout period has passed or the Web server is being stopped
Loaded
Created
Initialized
Served
Destroyed
Servlet is invoked for the first time
The same servlet is invoked again, bypassing the Loaded, Created, and Initialized states, as long as it has not been destroyed
Invokes the service method

FIGURE 37.11 The JVM uses the `init`, `service`, and `destroy` methods to control the servlet.

2. The service method is invoked each time the server receives a request for the servlet. The server spawns a new thread and invokes service.
3. The destroy method is invoked after a timeout period has passed or as the Web server is terminated. This method releases resources for the servlet.

37.4.2 The GenericServlet Class, ServletConfig Interface, and HttpServlet Class

The javax.servlet.GenericServlet class defines a generic, protocol-independent servlet. It implements javax.servlet.Servlet and javax.servlet.ServletConfig. ServletConfig is an interface that defines four methods (getInitParameter, getInitParameterNames, getServletContext, and getServletName) for obtaining information from a Web server during initialization. All the methods in Servlet and ServletConfig are implemented in GenericServlet except service. Therefore, GenericServlet is an abstract class.

The javax.servlet.http.HttpServlet class defines a servlet for the HTTP protocol. It extends GenericServlet and implements the service method. The service method is implemented as a dispatcher of HTTP requests. The HTTP requests are processed in the following methods:

- doGet is invoked to respond to a GET request.
- doPost is invoked to respond to a POST request.
- doDelete is invoked to respond to a DELETE request. Such a request is normally used to delete a file on the server.
- doPut is invoked to respond to a PUT request. Such a request is normally used to send a file to the server.
- doOptions is invoked to respond to an OPTIONS request. This returns information about the server, such as which HTTP methods it supports.
- doTrace is invoked to respond to a TRACE request. Such a request is normally used for debugging. This method returns an HTML page that contains appropriate trace information.

All these methods use the following signature:

```
protected void doXxx(HttpServletRequest req, HttpServletResponse resp)
  throws ServletException, java.io.IOException
```

The HttpServlet class provides default implementation for these methods. You need to override doGet, doPost, doDelete, and doPut if you want the servlet to process a GET, POST, DELETE, or PUT request. By default, nothing will be done. Normally, you should not override the doOptions method unless the servlet implements new HTTP methods beyond those implemented by HTTP 1.1. Nor is there any need to override the doTrace method.

Note
GET and POST requests are often used, whereas DELETE, PUT, OPTIONS, and TRACE are not. For more information about these requests, please refer to the HTTP 1.1 specification from www.cis.ohio-state.edu/htbin/rfc/rfc2068.html.

Note
Although the methods in HttpServlet are all nonabstract, HttpServlet is defined as an abstract class. Thus you cannot create a servlet directly from HttpServlet. Instead you have to define your servlet by extending HttpServlet.

The relationship of these interfaces and classes is shown in Figure 37.12.

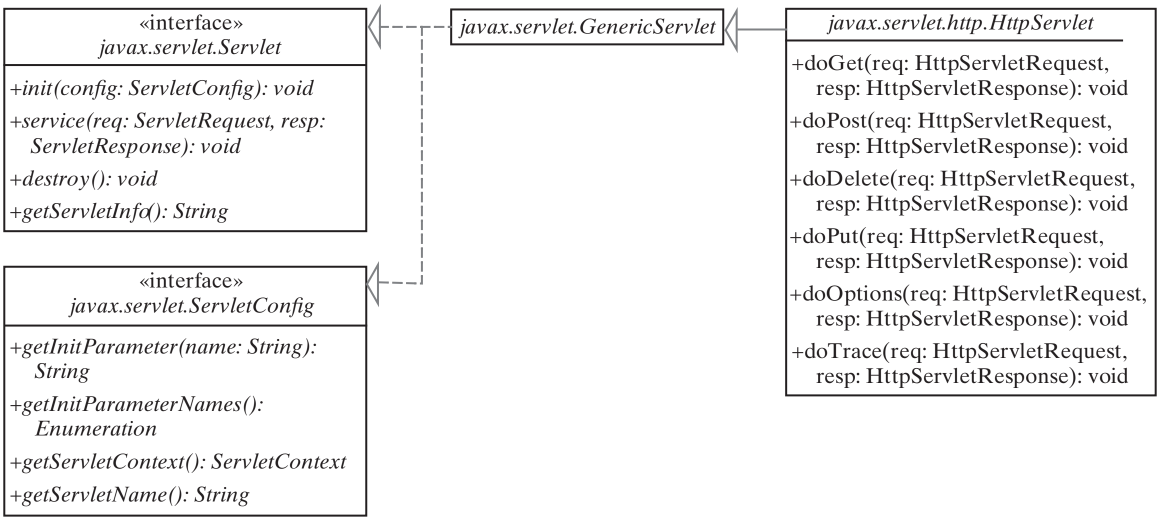

«interface»
javax.servlet.Servlet
+init(config: ServletConfig): void
+service(req: ServletRequest, resp: ServletResponse): void
+destroy(): void
+getServletInfo(): String
«interface»
javax.servlet.ServletConfig
+getInitParameter(name: String): String
+getInitParameterNames(): Enumeration
+getServletContext(): ServletContext
+getServletName(): String
javax.servlet.GenericServlet
javax.servlet.http.HttpServlet
+doGet(req: HttpServletRequest, resp: HttpServletResponse): void
+doPost(req: HttpServletRequest, resp: HttpServletResponse): void
+doDelete(req: HttpServletRequest, resp: HttpServletResponse): void
+doPut(req: HttpServletRequest, resp: HttpServletResponse): void
+doOptions(req: HttpServletRequest, resp: HttpServletResponse): void
+doTrace(req: HttpServletRequest, resp: HttpServletResponse): void

FIGURE 37.12 `HttpServlet` inherits abstract class `GenericServlet`, which implements interfaces `Servlet` and `ServletConfig`.

37.4.3 The `ServletRequest` Interface and `HttpServletRequest` Interface

Every `doXxx` method in the `HttpServlet` class has a parameter of the `HttpServletRequest` type, which is an object that contains HTTP request information, including parameter name and values, attributes, and an input stream. `HttpServletRequest` is a subinterface of `ServletRequest`. `ServletRequest` defines a more general interface to provide information for all kinds of clients. The frequently used methods in these two interfaces are shown in Figure 37.13.

37.4.4 The `ServletResponse` Interface and `HttpServletResponse` Interface

Every `doXxx` method in the `HttpServlet` class has a parameter of the `HttpServletResponse` type, which is an object that assists a servlet in sending a response to the client. `HttpServletResponse` is a subinterface of `ServletResponse`. `ServletResponse` defines a more general interface for sending output to the client.

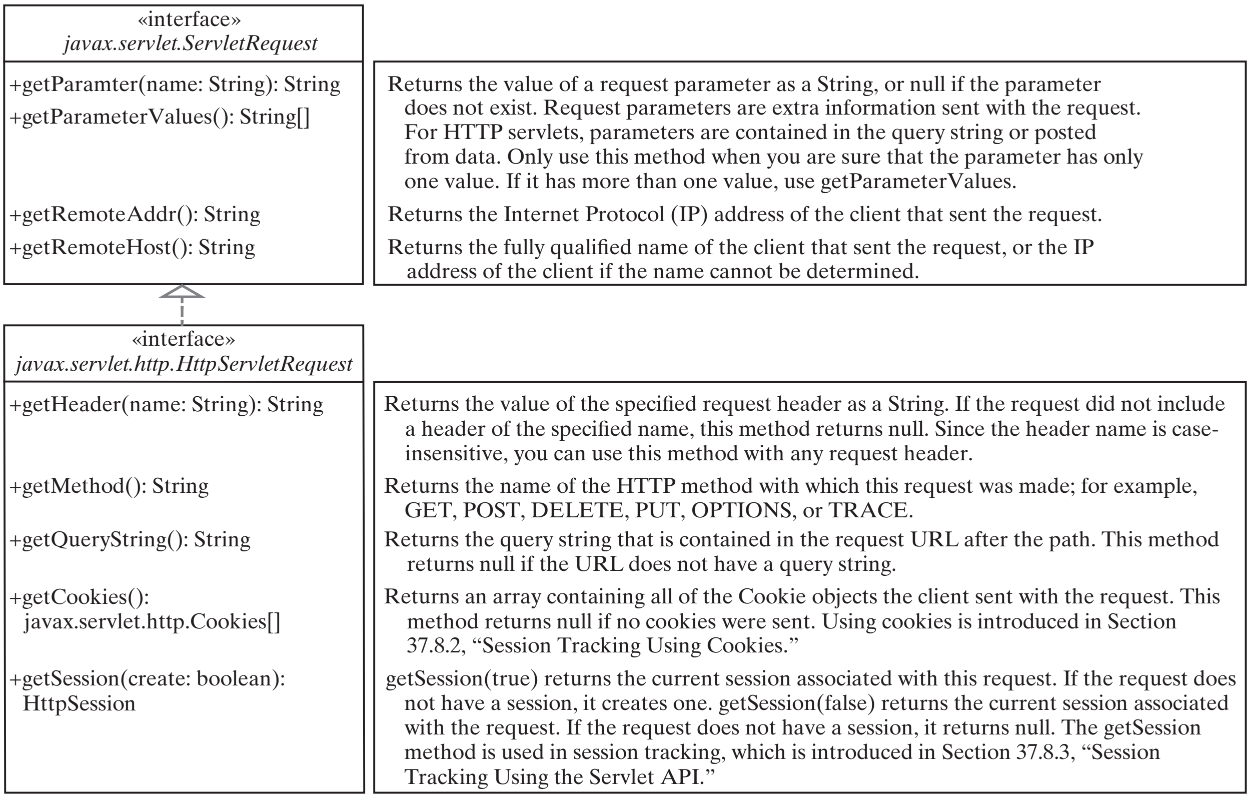
The frequently used methods in these two interfaces are shown in Figure 37.14.

Check Point

37.4.1 Describe the life cycle of a servlet.

37.4.2 Suppose you started the Web server, ran the following servlet twice by issuing an appropriate URL from the Web browser, and finally stopped Tomcat. What was displayed on the console when the servlet was first invoked? What was displayed on the console when the servlet was invoked for the second time? What was displayed on the console when Tomcat was shut down?

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class Test extends HttpServlet {
  public Test() {
    System.out.println("Constructor called");
  }

  /** Initialize variables */
  public void init() throws ServletException {
      System.out.println("init called");
    }

    /** Process the HTTP Get request */
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
          throws ServletException, IOException {
      System.out.println("doGet called");
    }

    /** Clean up resources */
    public void destroy() {
      System.out.println("destroy called");
    }
  }
```


«interface»
javax.servlet.ServletRequest
+getParamter(name: String): String
+getParameterValues(): String[]
+getRemoteAddr(): String
+getRemoteHost(): String
«interface»
javax.servlet.http.HttpServletRequest
+getHeader(name: String): String
+getMethod(): String
+getQueryString(): String
+getCookies(): javax.servlet.http.Cookies[]
+getSession(create: boolean): HttpSession

Returns the value of a request parameter as a String, or null if the parameter does not exist. Request parameters are extra information sent with the request. For HTTP servlets, parameters are contained in the query string or posted from data. Only use this method when you are sure that the parameter has only one value. If it has more than one value, use getParameterValues.

Returns the Internet Protocol (IP) address of the client that sent the request.

Returns the fully qualified name of the client that sent the request, or the IP address of the client if the name cannot be determined.

Returns the value of the specified request header as a String. If the request did not include a header of the specified name, this method returns null. Since the header name is case-insensitive, you can use this method with any request header.

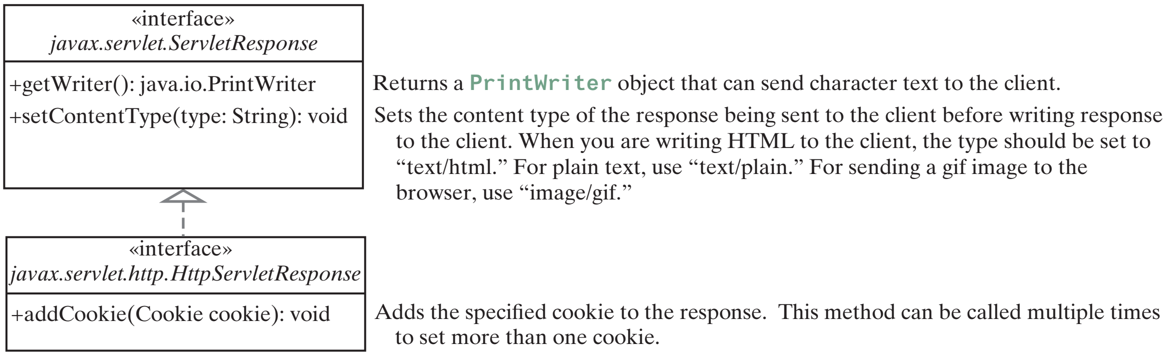
Returns the name of the HTTP method with which this request was made; for example, GET, POST, DELETE, PUT, OPTIONS, or TRACE.

Returns the query string that is contained in the request URL after the path. This method returns null if the URL does not have a query string.

Returns an array containing all of the Cookie objects the client sent with the request. This method returns null if no cookies were sent. Using cookies is introduced in Section 37.8.2, "Session Tracking Using Cookies."

getSession(true) returns the current session associated with this request. If the request does not have a session, it creates one. getSession(false) returns the current session associated with the request. If the request does not have a session, it returns null. The getSession method is used in session tracking, which is introduced in Section 37.8.3, "Session Tracking Using the Servlet API."

Figure 37.13 `HttpServletRequest` is a subinterface of `ServletRequest`.


«interface»
javax.servlet.ServletResponse
+getWriter(): java.io.PrintWriter
+setContentType(type: String): void
«interface»
javax.servlet.http.HttpServletResponse
+addCookie(Cookie cookie): void

Returns a `PrintWriter` object that can send character text to the client.

Sets the content type of the response being sent to the client before writing response to the client. When you are writing HTML to the client, the type should be set to "text/html." For plain text, use "text/plain." For sending a gif image to the browser, use "image/gif."

Adds the specified cookie to the response. This method can be called multiple times to set more than one cookie.

Figure 37.14 `HttpServletResponse` is a subinterface of `ServletResponse`.

37.5 Creating Servlets

Key Point

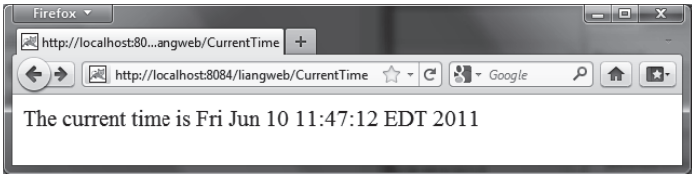
You can define a servlet class by extending the **HttpServlet** *class and implement the* **doGet** *and* **doPost** *methods.*

Servlets are the opposite of Java applets. Java applets run from a Web browser on the client side. To write Java programs, you define classes. To write a Java applet, you define a class that extends the Applet class. The Web browser runs and controls the execution of the applet through the methods defined in the Applet class. Similarly, to write a Java servlet, you define a class that extends the HttpServlet class. The servlet container runs and controls the execution of the servlet through the methods defined in the HttpServlet class. Like a Java applet, a servlet does not have a main method. A servlet depends on the servlet engine to call the methods. Every servlet has a structure like the one shown below:

```
package chapter37;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class MyServlet extends HttpServlet {
  /** Called by the servlet engine to initialize servlet */
  public void init() throws ServletException {
    ...
  }

  /** Process the HTTP Get request */
  public void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
    ...
  }

  /** Process the HTTP Post request */
  public void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
    ...
  }

  /** Called by the servlet engine to release resource */
  public void destroy() {
    ...
  }

  // Other methods if necessary
}
```

The servlet engine controls the servlets using init, doGet, doPost, destroy, and other methods. By default, the doGet and doPost methods do nothing. To handle a GET request, you need to override the doGet method; to handle a POST request, you need to override the doPost method.

Listing 37.2 gives a simple Java servlet that generates a dynamic webpage for displaying the current time, as shown in Figure 37.15.


Firefox
http://localhost:80...angweb/CurrentTime
http://localhost:8084/liangweb/CurrentTime
Google

The current time is Fri Jun 10 11:47:12 EDT 2011

FIGURE 37.15 Servlet CurrentTime displays the current time.

LISTING 37.2 CurrentTime.java

```java
package chapter37;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class CurrentTime extends HttpServlet {
  /** Process the HTTP Get request */
  public void doGet(HttpServletRequest request, HttpServletResponse
      response)  throws ServletException, IOException {
      response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<p>The current time is " + new java.util.Date());
    out.close(); // Close stream
  }
}
```

The `HttpServlet` class has a `doGet` method. The `doGet` method is invoked when the browser issues a request to the servlet using the GET method. Your servlet class should override the `doGet` method to respond to the GET request. In this case, you write the code to display the current time.

Servlets return responses to the browser through an `HttpServletResponse` object. Since the `setContentType("text/html")` method sets the MIME type to "text/html," the browser will display the response in HTML. The `getWriter` method returns a `PrintWriter` object for sending HTML back to the client.

Note

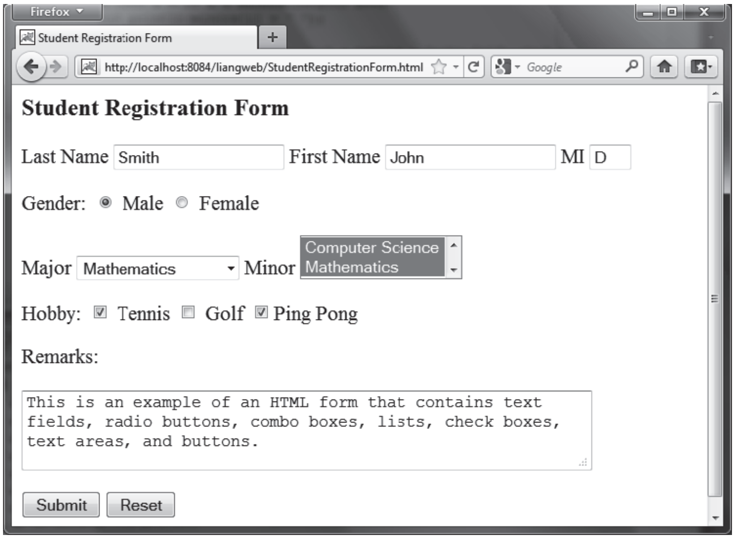
The URL query string uses the GET method to issue a request to the servlet. The current time may not be current if the webpage for displaying the current time is cached. To ensure that a new current time is displayed, refresh the page in the browser. In the next example, you will write a new servlet that uses the POST method to obtain the current time.

37.6 HTML Forms

Key Point

HTML forms are used to collect and submit data from a client to a Web server.

HTML forms enable you to submit data to the Web server in a convenient form. As shown in Figure 37.16, the form can contain text fields, text area, check boxes, combo boxes, lists, radio buttons, and buttons.


Firefox
Student Registration Form
http://localhost:8084/liangweb/StudentRegistrationForm.html
Google
Submit
Reset

Student Registration Form

Last Name Smith First Name John MI D

Gender: ● Male ○ Female

Major Mathematics Minor Computer Science / Mathematics

Hobby: ☑ Tennis □ Golf ☑ Ping Pong

Remarks:

```
This is an example of an HTML form that contains text
fields, radio buttons, combo boxes, lists, check boxes,
text areas, and buttons.
```

FIGURE 37.16 An HTML form may contain text fields, radio buttons, combo boxes, lists, check boxes, text areas, and buttons.

The HTML code for creating the form in Figure 37.16 is given in Listing 37.3. (If you are unfamiliar with HTML, please see Supplement V.A, “HTML and XHTML Tutorial.”)

Listing 37.3 StudentRegistrationForm.html

```
<!--An HTML Form Demo -->
<html>
  <head>
    <title>Student Registration Form</title>
  </head>
  <body>
    <h3>Student Registration Form</h3>

    <form action = "GetParameters"
      method = "get">
      <!-- Name text fields -->
      <p><label>Last Name</label>
      <input type = "text" name = "lastName" size = "20" />
      <label>First Name</label>
      <input type = "text" name = "firstName" size = "20" />
      <label>MI</label>
      <input type = "text" name = "mi" size = "1" /></p>

      <!-- Gender radio buttons -->
      <p><label>Gender:</label>
      <input type = "radio" name = "gender" value = "M" checked />
      Male
      <input type = "radio" name = "gender" value = "F" /> Female</p>

      <!-- Major combo box -->
      <p><label>Major</label>
        <select name = "major" size = "1">
          <option value = "CS">Computer Science</option>
          <option value = "Math">Mathematics</option>
          <option>English</option>
          <option>Chinese</option>
        </select>

      <!-- Minor list -->
      <label>Minor</label>
        <select name = "minor" size = "2" multiple>
          <option>Computer Science</option>
          <option>Mathematics</option>
          <option>English</option>
          <option>Chinese</option>
        </select></p>

      <!-- Hobby check boxes -->
      <p><label>Hobby:</label>
        <input type = "checkbox" name = "tennis" /> Tennis
        <input type = "checkbox" name = "golf" /> Golf
        <input type = "checkbox" name = "pingPong" checked />Ping Pong
      </p>

      <!-- Remark text area -->
      <p>Remarks:</p>
      <p><textarea name = "remarks" rows = "3" cols = "56">
        </textarea></p>

      <!-- Submit and Reset buttons -->
      <p><input type = "submit" value = "Submit" />
      <input type = "reset" value = "Reset" /></p>
    </form>
  </body>
</html>
```

The following HTML tags are used to construct HTML forms:

- `<form>` ... `</form>` defines a form body. The attributes for the `<form>` tag are `action` and `method`. The `action` attribute specifies the server program to be executed on the Web server when the form is submitted. The `method` attribute is either get or post.
- `<label>` ... `</label>` simply defines a label.
- `<input>` defines an input field. The attributes for this tag are `type`, `name`, `value`, `checked`, `size`, and `maxlength`. The type attribute specifies the input type. Possible types are `text` for a one-line text field, `radio` for a radio button, and `checkbox` for a check box. The `name` attribute gives a formal name for the attribute. This `name` attribute is used by the servlet program to retrieve its associated value. The names of the radio buttons in a group must be identical. The value attribute specifies a default value for a text field and text area. The `checked` attribute indicates whether a radio button or a check box is initially checked. The `size` attribute specifies the size of a text field, and the `maxlength` attribute specifies the maximum length of a text field.
- `<select>` ... `</select>` defines a combo box or a list. The attributes for this tag are `name`, `size`, and `multiple`. The `size` attribute specifies the number of rows visible in the list. The `multiple` attribute specifies that multiple values can be selected from a list. Set `size` to `1` and do not use a `multiple` for a combo box.
- `<option>` ... `</option>` defines a selection list within a `<select>` ... `</select>` tag. This tag may be used with the value attribute to specify a value for the selected option (e.g., `<option value = "CS">`Computer Science). If no value is specified, the selected option is the value.
- `<textarea>` ... `</textarea>` defines a text area. The attributes are `name`, `rows`, and `cols`. The `rows` and `cols` attributes specify the number of rows and columns in a text area.

Note

You can create the HTML file from NetBeans. Right-click `liangweb` and choose *New, HTML*, to display the New HTML File dialog box. Enter `StudentRegistrationForm` as the file name and click *Finish* to create the file.

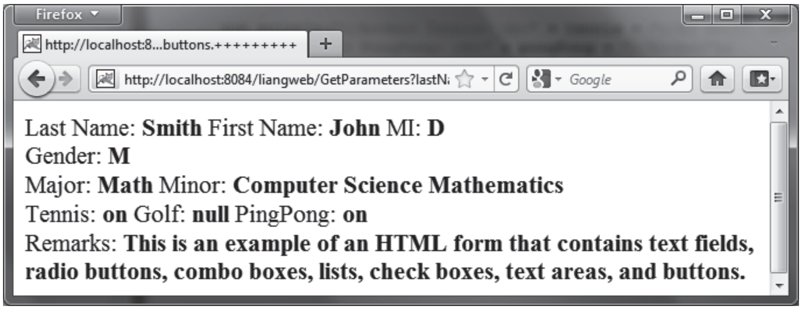
37.6.1 Obtaining Parameter Values from HTML Forms

To demonstrate how to obtain parameter values from an HTML form, Listing 37.4 creates a servlet to obtain all the parameter values from the preceding student registration form in Figure 37.16 and display their values, as shown in Figure 37.17.

Listing 37.4 GetParameters.java

```
package chapter37;

import javax.servlet.*;
```

```
 4  import javax.servlet.http.*;
 5  import java.io.*;
 6
 7  public class GetParameters extends HttpServlet {
 8    /** Process the HTTP Post request */
 9    public void doGet(HttpServletRequest request, HttpServletResponse
10        response) throws ServletException, IOException {
11      response.setContentType("text/html");
12      PrintWriter out = response.getWriter();
13
14      // Obtain parameters from the client
15      String lastName = request.getParameter("lastName");
16      String firstName = request.getParameter("firstName");
17      String mi = request.getParameter("mi");
18      String gender = request.getParameter("gender");
19      String major = request.getParameter("major");
20      String[] minors = request.getParameterValues("minor");
21      String tennis = request.getParameter("tennis");
22      String golf = request.getParameter("golf");
23      String pingPong = request.getParameter("pingPong");
24      String remarks = request.getParameter("remarks");
25
26      out.println("Last Name: <b>" + lastName + "</b> First Name: <b>"
27        + firstName + "</b> MI: <b>" + mi + "</b><br>");
28      out.println("Gender: <b>" + gender + "</b><br>");
29      out.println("Major: <b>" + major + "</b> Minor: <b>");
30
31      if (minors != null)
32        for (int i = 0; i < minors.length; i++)
33          out.println(minors[i] + " ");
34
35      out.println("</b><br> Tennis: <b>" + tennis + "</b> Golf: <b>" +
36        golf + "</b> PingPong: <b>" + pingPong + "</b><br>");
37      out.println("Remarks: <b>" + remarks + "</b>");
38      out.close(); // Close stream
39    }
40  }
```


Firefox
http://localhost:8...buttons.+++++++++
http://localhost:8084/liangweb/GetParameters?lastN
Google

Last Name: **Smith** First Name: **John** MI: **D**
Gender: **M**
Major: **Math** Minor: **Computer Science Mathematics**
Tennis: **on** Golf: **null** PingPong: **on**
Remarks: **This is an example of an HTML form that contains text fields, radio buttons, combo boxes, lists, check boxes, text areas, and buttons.**

FIGURE 37.17 The servlet displays the parameter values entered in Figure 37.16.

The HTML form is already created in StudentRegistrationForm.html and displayed in Figure 37.16. Since the action for the form is `GetParameters`, clicking the *Submit* button invokes the `GetParameters` servlet.

Each GUI component in the form has a name attribute. The servlet uses the name attribute in the `getParameter(attributeName)` method to obtain the parameter value as a string. In case of a list with multiple values, use the `getParameterValues(attributeName)` method to return the parameter values in an array of strings (line 20).

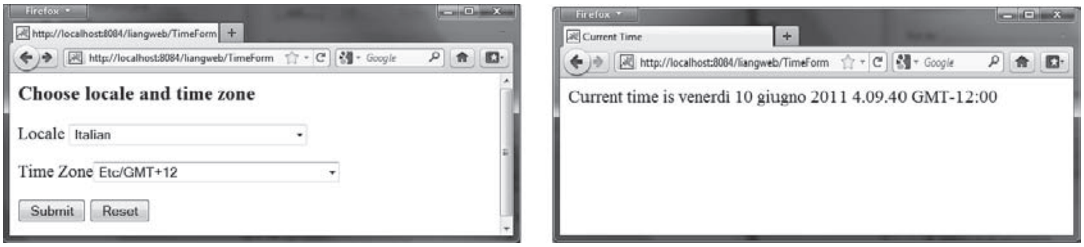
You may optionally specify the value attribute in a text field, text area, combo box, list, check box, or radio button in an HTML form. For text field and text area, the value attribute specifies a default value to be displayed in the text field and text area. The user can type in new values to replace it. For combo box, list, check box, and radio button, the value attribute specifies the parameter value to be returned from the getParameter and getParameterValues methods. If the value attribute is not specified for a combo box or a list, it returns the selected string from the combo box or the list. If the value attribute is not specified for a radio button or a check box, it returns string **on** for a checked radio button or a checked check box, and returns null for an unchecked check box.

Note

If an attribute does not exist, the getParameter(attributeName) method returns null. If an empty value of the parameter is passed to the servlet, the getParameter(attributeName) method returns a string with an empty value. In this case, the length of the string is 0.

37.6.2 Obtaining Current Time Based on Locale and Time Zone

This example creates a servlet that processes the GET and POST requests. The GET request generates a form that contains a combo box for locale and a combo box for time zone, as shown in Figure 37.18a. The user can choose a locale and a time zone from this form to submit a POST request to obtain the current time based on the locale and time zone, as shown in Figure 37.18b.



http://localhost:8084/liangweb/TimeForm

Choose locale and time zone

Locale Italian

Time Zone Etc/GMT+12

(a)

Current Time

http://localhost:8084/liangweb/TimeForm

Current time is venerdì 10 giugno 2011 4.09.40 GMT-12:00

(b)

FIGURE 37.18 The GET method in the TimeForm servlet displays a form in (a), and the POST method in the TimeForm servlet displays the time based on locale and time zone in (b).

Listing 37.5 gives the servlet.

LISTING 37.5 TimeForm.java

```
package chapter37;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import java.text.*;

public class TimeForm extends HttpServlet {
  private static final String CONTENT_TYPE = "text/html";
  private Locale[] allLocale = Locale.getAvailableLocales();
  private String[] allTimeZone = TimeZone.getAvailableIDs();

  /** Process the HTTP Get request */
  public void doGet(HttpServletRequest request, HttpServletResponse
16        response) throws ServletException, IOException {
17      response.setContentType(CONTENT_TYPE);
18      PrintWriter out = response.getWriter();
19      out.println("<h3>Choose locale and time zone</h3>");
20      out.println("<form method=\"post\" action=" +
21        "TimeForm>");
22      out.println("Locale <select size=\"1\" name=\"locale\">");
23
24      // Fill in all locales
25      for (int i = 0; i < allLocale.length; i++) {
26        out.println("<option value=\"" + i +"\">" +
27          allLocale[i].getDisplayName() + "</option>");
28      }
29      out.println("</select>");
30
31      // Fill in all time zones
32      out.println("<p>Time Zone<select size=\"1\" name=\"timezone\">");
33      for (int i = 0; i < allTimeZone.length; i++) {
34        out.println("<option value=\"" + allTimeZone[i] +"\">" +
35          allTimeZone[i] + "</option>");
36      }
37      out.println("</select>");
38
39      out.println("<p><input type=\"submit\" value=\"Submit\" >");
40      out.println("<input type=\"reset\" value=\"Reset\"></p>");
41      out.println("</form>");
42      out.close(); // Close stream
43    }
44
45    /** Process the HTTP Post request */
46    public void doPost(HttpServletRequest request, HttpServletResponse
47        response) throws ServletException, IOException {
48      response.setContentType(CONTENT_TYPE);
49      PrintWriter out = response.getWriter();
50      out.println("<html>");
51      int localeIndex = Integer.parseInt(
52        request.getParameter("locale"));
53      String timeZoneID = request.getParameter("timezone");
54      out.println("<head><title>Current Time</title></head>");
55      out.println("<body>");
56      Calendar calendar =
57        new GregorianCalendar(allLocale[localeIndex]);
58      TimeZone timeZone = TimeZone.getTimeZone(timeZoneID);
59      DateFormat dateFormat = DateFormat.getDateTimeInstance(
60        DateFormat.FULL, DateFormat.FULL, allLocale[localeIndex]);
61      dateFormat.setTimeZone(timeZone);
62      out.println("Current time is " +
63        dateFormat.format(calendar.getTime()) + "</p>");
64      out.println("</body></html>");
65      out.close(); // Close stream
66    }
67  }
```

When you run this servlet, the servlet `TimeForm`'s `doGet` method is invoked to generate the time form dynamically. The method of the form is POST, and the action invokes the same servlet, `TimeForm`. When the form is submitted to the server, the `doPost` method is invoked to process the request.

The variables `allLocale` and `allTimeZone` (lines 11–12), respectively, hold all the available locales and time zone IDs. The names of the locales are displayed in the locale list. The values for the locales are the indexes of the locales in the array `allLocale`. The time zone IDs are strings. They are displayed in the time zone list. They are also the values for the list. The indexes of the locale and the time zone are passed to the servlet as parameters. The **doPost** method obtains the values of the parameters (lines 51–53) and finds the current time based on the locale and time zone.

Note

If you choose an Asian locale (e.g., Chinese, Korean, or Japanese), the time will not be displayed properly, because the default character encoding is UTF-8. To fix this problem, insert the following statement in line 48 to set an international character encoding:

```
response.setCharacterEncoding("GB18030");
```

For information on encoding, see Section 36.6.6, "Character Encoding."

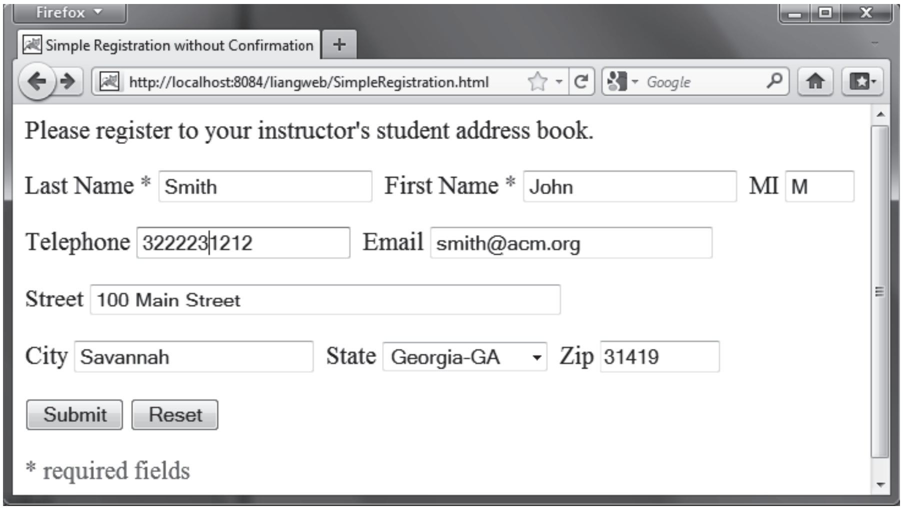
37.7 Database Programming in Servlets

Servlets can access and manipulate databases using JDBC.

Key Point

Many dynamic Web applications use databases to store and manage data. Servlets can connect to any relational database via JDBC. In Chapter 34, Java Database Programming, you learned how to create Java programs to access and manipulate relational databases via JDBC. Connecting a servlet to a database is no different from connecting a Java application or applet to a database. If you know Java servlets and JDBC, you can combine them to develop interesting and practical Web-based interactive projects.

To demonstrate connecting to a database from a servlet, let us create a servlet that processes a registration form. The client enters data in an HTML form and submits the form to the server, as shown in Figure 37.19. The result of the submission is shown in Figure 37.20. The server collects the data from the form and stores them in a database.


Firefox
Simple Registration without Confirmation
http://localhost:8084/liangweb/SimpleRegistration.html
Google

Please register to your instructor's student address book.

Last Name * Smith First Name * John MI M

Telephone 3222231212 Email smith@acm.org

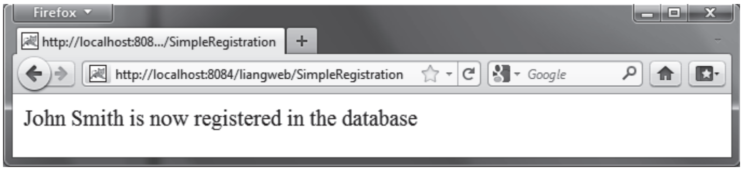
Street 100 Main Street

City Savannah State Georgia-GA Zip 31419

Submit Reset

* required fields

Figure 37.19 The HTML form enables the user to enter student information.


Firefox
http://localhost:808.../SimpleRegistration
http://localhost:8084/liangweb/SimpleRegistration
Google

John Smith is now registered in the database

Figure 37.20 The servlet processes the form and stores data in a database.

The registration data are stored in an Address table consisting of the following fields: firstName, mi, lastName, street, city, state, zip, telephone, and email, defined in the following statement:

```
create table Address (
  firstname varchar(25),
  mi char(1),
  lastname varchar(25),
  street varchar(40),
  city varchar(20),
  state varchar(2),
  zip varchar(5),
  telephone varchar(10),
  email varchar(30)
)
```

MySQL, Oracle, and Access were used in Chapter 34. You can use any relational database. If the servlet uses a database driver other than the JDBC-ODBC driver (e.g., the MySQL JDBC driver and the Oracle JDBC driver), you need to add the JDBC driver (e.g., mysqljdbc.jar for MySQL and ojdbc6.jar for Oracle) into the Libraries node in the project.

Create an HTML file named **SimpleRegistration.html** in Listing 37.6 for collecting the data and sending them to the database using the post method.

LISTING 37.6 SimpleRegistration.html

```
<!-- SimpleRegistration.html -->
<html>
  <head>
    <title>Simple Registration without Confirmation</title>
  </head>
  <body>
    Please register to your instructor’s student address book.

    <form method = "post" action = "SimpleRegistration">
      <p>Last Name <font color = "#FF0000">*</font>
        <input type = "text" name = "lastName">&nbsp;
        First Name <font color = "#FF0000">*</font>
        <input type = "text" name = "firstName">&nbsp;
        MI <input type = "text" name = "mi" size = "3">
      </p>
      <p>Telephone
        <input type = "text" name = "telephone" size = "20">&nbsp;
        Email
        <input type = "text" name = "email" size = "28">&nbsp;
      </p>
      <p>Street <input type = "text" name = "street" size = "50">
      </p>
      <p>City <input type = "text" name = "city" size = "23">&nbsp;
        State
        <select size = "1" name = "state">
          <option value = "GA">Georgia-GA</option>
          <option value = "OK">Oklahoma-OK</option>
          <option value = "IN">Indiana-IN</option>
        </select>&nbsp;
        Zip <input type = "text" name = "zip" size = "9">
      </p>
      <p><input type = "submit" name = "Submit" value = "Submit">
         <input type = "reset" value = "Reset">
      </p>
    </form>
```

```
      <p><font color = "#FF0000">* required fields</font></p>
    </body>
  </html>
```

Create the servlet named **SimpleRegistration** in Listing 37.7.

Listing 37.7 SimpleRegistration.java

```
package chapter37;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;

public class SimpleRegistration extends HttpServlet {
  // Use a prepared statement to store a student into the database
  private PreparedStatement pstmt;

  /** Initialize variables */
  public void init() throws ServletException {
    initializeJdbc();
  }

  /** Process the HTTP Post request */
  public void doPost(HttpServletRequest request, HttpServletResponse
      response) throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    // Obtain parameters from the client
    String lastName = request.getParameter("lastName");
    String firstName = request.getParameter("firstName");
    String mi = request.getParameter("mi");
    String phone = request.getParameter("telephone");
    String email = request.getParameter("email");
    String address = request.getParameter("street");
    String city = request.getParameter("city");
    String state = request.getParameter("state");
    String zip = request.getParameter("zip");

    try {
      if (lastName.length() == 0 || firstName.length() == 0) {
        out.println("Last Name and First Name are required");
      }
      else {
        storeStudent(lastName, firstName, mi, phone, email,
          address, city, state, zip);

        out.println(firstName + " " + lastName +
          " is now registered in the database");
      }
    }
    catch(Exception ex) {
      out.println("Error: " + ex.getMessage());
    }
    finally {
      out.close(); // Close stream
    }
  }

54    /** Initialize database connection */
55    private void initializeJdbc() {
56      try {
57        // Load the JDBC driver
58        Class.forName("com.mysql.jdbc.Driver");
59        System.out.println("Driver loaded");
60
61        // Establish a connection
62        Connection conn = DriverManager.getConnection
63          ("jdbc:mysql://localhost/javabook", "scott", "tiger");
64        System.out.println("Database connected");
65
66        // Create a Statement
67        pstmt = conn.prepareStatement("insert into Address " +
68          "(lastName, firstName, mi, telephone, email, street, city, "
69          + "state, zip) values (?, ?, ?, ?, ?, ?, ?, ?, ?)");
70      }
71      catch (Exception ex) {
72        ex.printStackTrace();
73      }
74    }
75
76    /** Store a student record to the database */
77    private void storeStudent(String lastName, String firstName,
78        String mi, String phone, String email, String address,
79        String city, String state, String zip) throws SQLException {
80      pstmt.setString(1, lastName);
81      pstmt.setString(2, firstName);
82      pstmt.setString(3, mi);
83      pstmt.setString(4, phone);
84      pstmt.setString(5, email);
85      pstmt.setString(6, address);
86      pstmt.setString(7, city);
87      pstmt.setString(8, state);
88      pstmt.setString(9, zip);
89      pstmt.executeUpdate();
90    }
91  }
```

The `init` method (line 13) is executed once when the servlet starts. After the servlet has started, the servlet can be invoked many times as long as it is alive in the servlet container. Load the driver and connect to the database from the servlet's `init` method (line 14). If a prepared statement or a callable statement is used, it should also be created in the `init` method. In this example, a prepared statement is desirable, because the servlet always uses the same insert statement with different values.

A servlet can connect to any relational database via JDBC. The `initializeJdbc` method in this example connects to a MySQL database (line 58). Once connected, it creates a prepared statement for inserting a student record into the database. MySQL is used in this example; you can replace it with any relational database.

Last name and first name are required fields. If either of them is empty, the servlet sends an error message to the client (lines 35–36). Otherwise, the servlet stores the data in the database using the prepared statement.

Check Point

37.7.1 What would be displayed if you changed the content type to `html/plain` in Listing 37.2, CurrentTime.java?

37.7.2 The statement `out.close()` is used to close the output stream to response. Why isn't this statement enclosed in a try-catch block?

37.7.3 What happens when you invoke `request.getParameter(paramName)` if `paramName` does not exist?

37.7.4 How do you write a text field, combo box, check box, and text area in an HTML form?

37.7.5 How do you retrieve the parameter value for a text field, combo box, list, check box, radio button, and text area from an HTML form?

37.7.6 If the servlet uses a database driver other than the JDBC-ODBC driver, where should the driver be placed in NetBeans?

37.8 Session Tracking

You can perform session tracking using hidden values in a form, using cookies, or using `HttpSession`*.*

Key Point

Web servers use the Hyper-Text Transport Protocol (HTTP). HTTP is a stateless protocol. An HTTP Web server cannot associate requests from a client, and therefore treats each request independently. This protocol works fine for simple Web browsing, where each request typically results in an HTML file or a text file being sent back to the client. Such simple requests are isolated. However, the requests in interactive Web applications are often related. Consider the two requests in the following scenario:

> Request 1: A client sends registration data to the server; the server then returns the data to the user for confirmation.
>
> Request 2: The client confirms the data that was submitted in Request 1.

In Request 2, the data submitted in Request 1 are confirmed. These two requests are related in a session. A *session* can be defined as a series of related interactions between a single client and the Web server over a period of time. Tracking data among requests in a session is known as *session tracking*.

This section introduces three techniques for session tracking: using hidden values, using cookies, and using the session tracking tools from servlet API.

37.8.1 Session Tracking Using Hidden Values

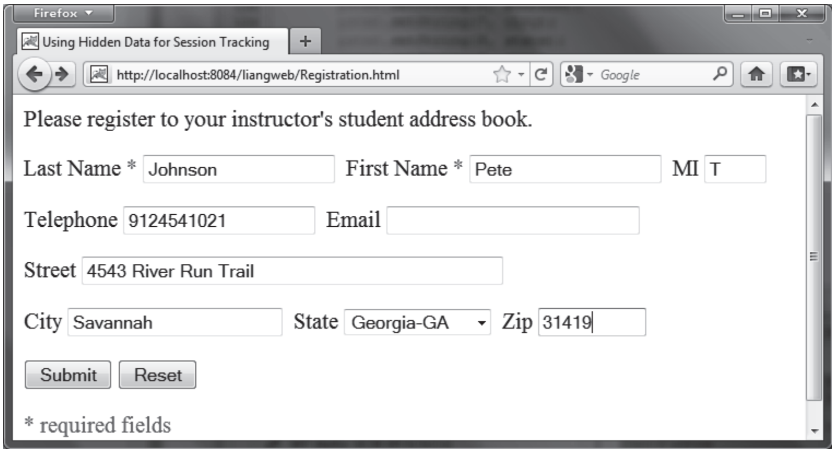
You can track a session by passing data from the servlet to the client as hidden values in a dynamically generated HTML form by including a field like this one:

```
<input type = "hidden" name = "lastName" value = "Smith">
```

The next request will submit the data back to the servlet. The servlet retrieves this hidden value just like any other parameter value, using the `getParameter` method.

Let us use an example to demonstrate using hidden values in a form. The example creates a servlet that processes a registration form. The client submits the form using the GET method, as shown in Figure 37.21. The server collects the data in the form, displays them to the client, and asks the client for confirmation, as shown in Figure 37.22. The client confirms the data by submitting the request with the hidden values using the POST method. Finally, the servlet writes the data to a database.

Create an HTML form named Registration.html in Listing 37.8 for collecting the data and sending it to the database using the GET method for confirmation. This file is almost identical to Listing 37.6, **SimpleRegistration.html** except that the action is replaced by `Registration` (line 9).



Firefox
Using Hidden Data for Session Tracking
http://localhost:8084/liangweb/Registration.html
Google

Please register to your instructor's student address book.

Last Name * Johnson First Name * Pete MI T

Telephone 9124541021 Email

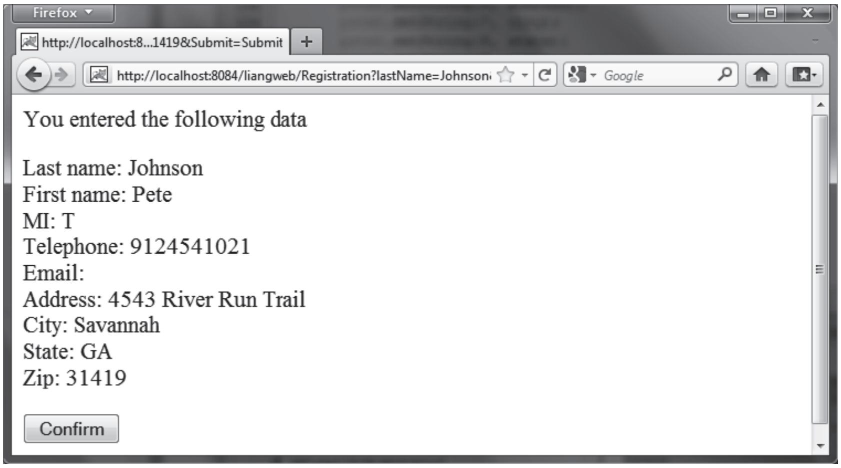
Street 4543 River Run Trail

City Savannah State Georgia-GA Zip 31419

Submit Reset

* required fields

FIGURE 37.21 The registration form collects user information.



Firefox
http://localhost:8...14198&Submit=Submit
http://localhost:8084/liangweb/Registration?lastName=Johnson
Google

You entered the following data

Last name: Johnson
First name: Pete
MI: T
Telephone: 9124541021
Email:
Address: 4543 River Run Trail
City: Savannah
State: GA
Zip: 31419

Confirm

FIGURE 37.22 The servlet asks the client for confirmation of the input.

LISTING 37.8 Registration.html

```
<!-- Registration.html -->
<html>
  <head>
    <title>Using Hidden Data for Session Tracking</title>
  </head>
  <body>
    Please register to your instructor’s student address book.

    <form method = "get" action = "Registration">
      <p>Last Name <font color = "#FF0000">*</font>
        <input type = "text" name = "lastName"> &nbsp;
        First Name <font color = "#FF0000">*</font>
        <input type = "text" name = "firstName"> &nbsp;
        MI  <input type = "text" name = "mi" size = "3">
      </p>
      <p>Telephone
        <input type = "text" name = "telephone" size = "20"> &nbsp;
        Email
        <input type = "text" name = "email" size = "28"> &nbsp;
      </p>
      <p>Street <input type = "text" name = "street" size = "50">
      </p>
      <p>City <input type = "text" name = "city" size = "23"> &nbsp;
        State
        <select size = "1" name = "state">
          <option value = "GA">Georgia-GA</option>
          <option value = "OK">Oklahoma-OK</option>
          <option value = "IN">Indiana-IN</option>
        </select> &nbsp;
        Zip <input type = "text" name = "zip" size = "9">
      </p>
      <p><input type = "submit" name = "Submit" value = "Submit">
         <input type = "reset" value = "Reset">
      </p>
    </form>
    <p><font color = "#FF0000">* required fields</font></p>
  </body>
</html>
```

Create the servlet named Registration in Listing 37.9.

Listing 37.9 Registration.java

```
package chapter37;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;

public class Registration extends HttpServlet {
  // Use a prepared statement to store a student into the database
  private PreparedStatement pstmt;

  /** Initialize variables */
  public void init() throws ServletException {
    initializeJdbc();
  }

  /** Process the HTTP Get request */
  public void doGet(HttpServletRequest request, HttpServletResponse
      response) throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    // Obtain data from the form
    String lastName = request.getParameter("lastName");
    String firstName = request.getParameter("firstName");
    String mi = request.getParameter("mi");
    String telephone = request.getParameter("telephone");
    String email = request.getParameter("email");
    String street = request.getParameter("street");
    String city = request.getParameter("city");
    String state = request.getParameter("state");
    String zip = request.getParameter("zip");

    if (lastName.length() == 0 || firstName.length() == 0) {
      out.println("Last Name and First Name are required");
    }
    else {
      // Ask for confirmation
      out.println("You entered the following data");
      out.println("<p>Last name: " + lastName);
      out.println("<br>First name: " + firstName);
      out.println("<br>MI: " + mi);
      out.println("<br>Telephone: " + telephone);
      out.println("<br>Email: " + email);
      out.println("<br>Address: " + street);
      out.println("<br>City: " + city);
      out.println("<br>State: " + state);
      out.println("<br>Zip: " + zip);

      // Set the action for processing the answers
      out.println("<p><form method=\"post\" action=" +
        "Registration>");
      // Set hidden values
      out.println("<p><input type=\"hidden\" " +
        "value=" + lastName + " name=\"lastName\">");
      out.println("<p><input type=\"hidden\" " +
        "value=" + firstName + " name=\"firstName\">");
      out.println("<p><input type=\"hidden\" " +
        "value=" + mi + " name=\"mi\">");
      out.println("<p><input type=\"hidden\" " +
        "value=" + telephone + " name=\"telephone\">");
      out.println("<p><input type=\"hidden\" " +
        "value=" + email + " name=\"email\">");
      out.println("<p><input type=\"hidden\" " +
        "value=" + street + " name=\"street\">");
      out.println("<p><input type=\"hidden\" " +
        "value=" + city + " name=\"city\">");
      out.println("<p><input type=\"hidden\" " +
        "value=" + state + " name=\"state\">");
      out.println("<p><input type=\"hidden\" " +
        "value=" + zip + " name=\"zip\">");
      out.println("<p><input type=\"submit\" value=\"Confirm\" >");
      out.println("</form>");
    }

    out.close(); // Close stream
  }

  /** Process the HTTP Post request */
  public void doPost(HttpServletRequest request, HttpServletResponse
      response) throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    try {
      String lastName = request.getParameter("lastName");
      String firstName = request.getParameter("firstName");
      String mi = request.getParameter("mi");
      String telephone = request.getParameter("telephone");
      String email = request.getParameter("email");
      String street = request.getParameter("street");
      String city = request.getParameter("city");
      String state = request.getParameter("state");
      String zip = request.getParameter("zip");

      storeStudent(lastName, firstName, mi, telephone, email,
        street, city, state, zip);

      out.println(firstName + " " + lastName +
        " is now registered in the database");
    }
    catch(Exception ex) {
      out.println("Error: " + ex.getMessage());
    }
  }

  /** Initialize database connection */
  private void initializeJdbc() {
    try {
      // Load the JDBC driver
      Class.forName("com.mysql.jdbc.Driver");
      System.out.println("Driver loaded");

      // Establish a connection
      Connection conn = DriverManager.getConnection
        ("jdbc:mysql://localhost/javabook" , "scott", "tiger");
      System.out.println("Database connected");

      // Create a Statement
      pstmt = conn.prepareStatement("insert into Address " +
        "(lastName, firstName, mi, telephone, email, street, city, "
        + "state, zip) values (?, ?, ?, ?, ?, ?, ?, ?, ?)");
    }
    catch (Exception ex) {
      System.out.println(ex);
    }
  }

  /** Store a student record to the database */
  private void storeStudent(String lastName, String firstName,
      String mi, String phone, String email, String address,
      String city, String state, String zip) throws SQLException {
    pstmt.setString(1, lastName);
    pstmt.setString(2, firstName);
    pstmt.setString(3, mi);
    pstmt.setString(4, phone);
    pstmt.setString(5, email);
    pstmt.setString(6, address);
    pstmt.setString(7, city);
    pstmt.setString(8, state);
    pstmt.setString(9, zip);
    pstmt.executeUpdate();
  }
}
```

The servlet processes the GET request by generating an HTML page that displays the client's input and asks for the client's confirmation. The input data consist of hidden values in the newly generated forms, so they will be sent back in the confirmation request. The confirmation request uses the POST method. The servlet retrieves the hidden values and stores them in the database.

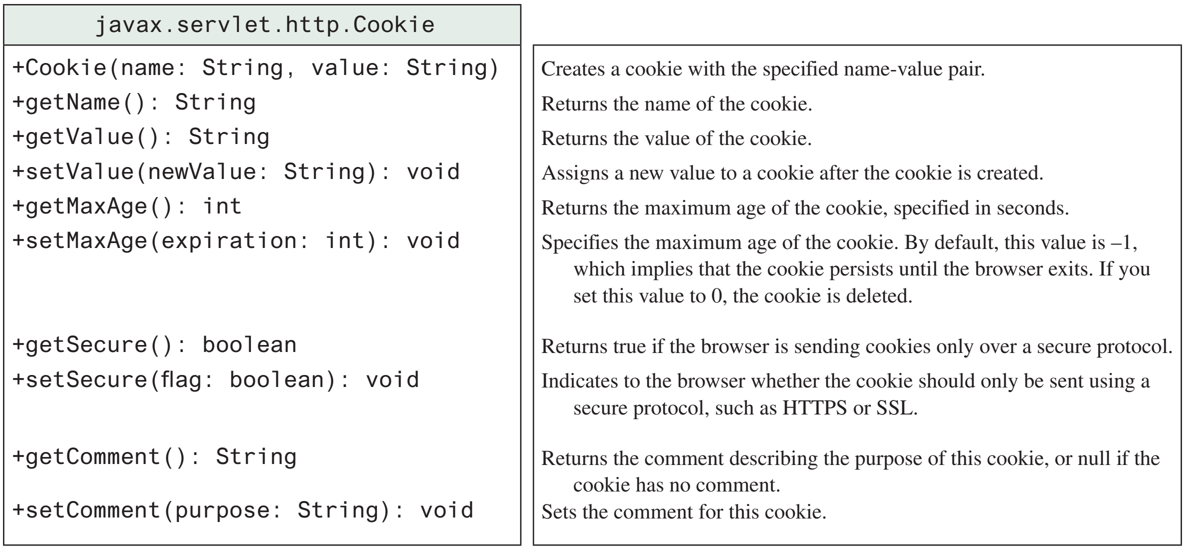
Since the first request does not write anything to the database, it is appropriate to use the GET method. Since the second request results in an update to the database, the POST method must be used.

Note

The hidden values could also be sent from the URL query string if the request used the GET method.

37.8.2 Session Tracking Using Cookies

You can track sessions using cookies, which are small text files that store sets of name/value pairs on the disk in the client's computer. Cookies are sent from the server through the instructions in the header of the HTTP response. The instructions tell the browser to create a cookie with a given name and its associated value. If the browser already has a cookie with the key name, the value will be updated. The browser will then send the cookie with any request submitted to the same server. Cookies can have expiration dates set, after which they will not be sent to the server. The `javax.servlet.http.Cookie` is used to create and manipulate cookies, as shown in Figure 37.23.



javax.servlet.http.Cookie

+Cookie(name: String, value: String) — Creates a cookie with the specified name-value pair.
+getName(): String — Returns the name of the cookie.
+getValue(): String — Returns the value of the cookie.
+setValue(newValue: String): void — Assigns a new value to a cookie after the cookie is created.
+getMaxAge(): int — Returns the maximum age of the cookie, specified in seconds.
+setMaxAge(expiration: int): void — Specifies the maximum age of the cookie. By default, this value is –1, which implies that the cookie persists until the browser exits. If you set this value to 0, the cookie is deleted.
+getSecure(): boolean — Returns true if the browser is sending cookies only over a secure protocol.
+setSecure(flag: boolean): void — Indicates to the browser whether the cookie should only be sent using a secure protocol, such as HTTPS or SSL.
+getComment(): String — Returns the comment describing the purpose of this cookie, or null if the cookie has no comment.
+setComment(purpose: String): void — Sets the comment for this cookie.

FIGURE 37.23 `Cookie` stores a name/value pair and other information about the cookie.

To send a cookie to the browser, use the `addCookie` method in the `HttpServletResponse` class, as shown below:

```
response.addCookie(cookie);
```

where `response` is an instance of `HttpServletResponse`.

To obtain cookies from a browser, use

```
request.getCookies();
```

where `request` is an instance of `HttpServletRequest`.

To demonstrate the use of cookies, let us create an example that accomplishes the same task as Listing 37.9, Registration.java. Instead of using hidden values for session tracking, it uses cookies.

Create the servlet named RegistrationWithCookie in Listing 37.10. Create an HTML file named RegistrationWithCookie.html that is identical to Registration.html except that the action is replaced by `RegistrationWithCookie.java`.

LISTING 37.10 `RegistrationWithCookie.java`

```
package chapter37;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;

public class RegistrationWithCookie extends HttpServlet {
  private static final String CONTENT_TYPE = "text/html";
  // Use a prepared statement to store a student into the database
  private PreparedStatement pstmt;

  /** Initialize variables */
  public void init() throws ServletException {
    initializeJdbc();
  }

  /** Process the HTTP Get request */
  public void doGet(HttpServletRequest request, HttpServletResponse
      response) throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    // Obtain data from the form
    String lastName = request.getParameter("lastName");
    String firstName = request.getParameter("firstName");
    String mi = request.getParameter("mi");
    String telephone = request.getParameter("telephone");
    String email = request.getParameter("email");
    String street = request.getParameter("street");
    String city = request.getParameter("city");
    String state = request.getParameter("state");
    String zip = request.getParameter("zip");

    if (lastName.length() == 0 || firstName.length() == 0) {
      out.println("Last Name and First Name are required");
    }
    else {
      // Create cookies and send cookies to browsers
      Cookie cookieLastName = new Cookie("lastName", lastName);
      // cookieLastName.setMaxAge(1000);
      response.addCookie(cookieLastName);
      Cookie cookieFirstName = new Cookie("firstName", firstName);
      response.addCookie(cookieFirstName);
      // cookieFirstName.setMaxAge(0);
      Cookie cookieMi = new Cookie("mi", mi);
      response.addCookie(cookieMi);
      Cookie cookieTelephone = new Cookie("telephone", telephone);
      response.addCookie(cookieTelephone);
      Cookie cookieEmail = new Cookie("email", email);
      response.addCookie(cookieEmail);
      Cookie cookieStreet = new Cookie("street", street);
      response.addCookie(cookieStreet);
      Cookie cookieCity = new Cookie("city", city);
      response.addCookie(cookieCity);
      Cookie cookieState = new Cookie("state", state);
      response.addCookie(cookieState);
      Cookie cookieZip = new Cookie("zip", zip);
      response.addCookie(cookieZip);

      // Ask for confirmation
      out.println("You entered the following data");
      out.println("<p>Last name: " + lastName);
      out.println("<br>First name: " + firstName);
      out.println("<br>MI: " + mi);
      out.println("<br>Telephone: " + telephone);
      out.println("<br>Email: " + email);
      out.println("<br>Street: " + street);
      out.println("<br>City: " + city);
      out.println("<br>State: " + state);
      out.println("<br>Zip: " + zip);

      // Set the action for processing the answers
      out.println("<p><form method=\"post\" action=" +
        "RegistrationWithCookie>");
      out.println("<p><input type=\"submit\" value=\"Confirm\" >");
      out.println("</form>");
    }

    out.close(); // Close stream
  }

  /** Process the HTTP Post request */
  public void doPost(HttpServletRequest request, HttpServletResponse
      response) throws ServletException, IOException {
    response.setContentType(CONTENT_TYPE);
    PrintWriter out = response.getWriter();

    String lastName = "";
    String firstName = "";
    String mi = "";
    String telephone = "";
    String email = "";
    String street = "";
    String city = "";
    String state = "";
    String zip = "";

    // Read the cookies
    Cookie[] cookies = request.getCookies();

    // Get cookie values
    for (int i = 0; i < cookies.length; i++) {
      if (cookies[i].getName().equals("lastName"))
        lastName = cookies[i].getValue();
      else if (cookies[i].getName().equals("firstName"))
        firstName = cookies[i].getValue();
      else if (cookies[i].getName().equals("mi"))
        mi = cookies[i].getValue();
      else if (cookies[i].getName().equals("telephone"))
        telephone = cookies[i].getValue();
      else if (cookies[i].getName().equals("email"))
        email = cookies[i].getValue();
      else if (cookies[i].getName().equals("street"))
        street = cookies[i].getValue();
      else if (cookies[i].getName().equals("city"))
        city = cookies[i].getValue();
      else if (cookies[i].getName().equals("state"))
        state = cookies[i].getValue();
      else if (cookies[i].getName().equals("zip"))
        zip = cookies[i].getValue();
    }

    try {
      storeStudent(lastName, firstName, mi, telephone, email, street,
        city, state, zip);

```

```
128       out.println(firstName + " " + lastName +
129         " is now registered in the database");
130
131       out.close(); // Close stream
132     }
133     catch(Exception ex) {
134       out.println("Error: " + ex.getMessage());
135     }
136   }
137
138   /** Initialize database connection */
139   private void initializeJdbc() {
140     try {
141       // Load the JDBC driver
142       Class.forName("com.mysql.jdbc.Driver");
143       System.out.println("Driver loaded");
144
145       // Establish a connection
146       Connection conn = DriverManager.getConnection
147         ("jdbc:mysql://localhost/javabook" , "scott", "tiger");
148       System.out.println("Database connected");
149
150       // Create a Statement
151       pstmt = conn.prepareStatement("insert into Address " +
152         "(lastName, firstName, mi, telephone, email, street, city, "
153         + "state, zip) values (?, ?, ?, ?, ?, ?, ?, ?, ?)");
154     }
155     catch (Exception ex) {
156       System.out.println(ex);
157     }
158   }
159
160   /** Store a student record to the database */
161   private void storeStudent(String lastName, String firstName,
162       String mi, String telephone, String email, String street,
163       String city, String state, String zip) throws SQLException {
164     pstmt.setString(1, lastName);
165     pstmt.setString(2, firstName);
166     pstmt.setString(3, mi);
167     pstmt.setString(4, telephone);
168     pstmt.setString(5, email);
169     pstmt.setString(6, street);
170     pstmt.setString(7, city);
171     pstmt.setString(8, state);
172     pstmt.setString(9, zip);
173     pstmt.executeUpdate();
174   }
175 }
```

You have to create a cookie for each value you want to track, using the Cookie class's only constructor, which defines a cookie's name and value as shown below (line 40):

```
Cookie cookieLastName = new Cookie("lastName", lastName);
```

To send the cookie to the browser, use a statement like this one (line 42):

```
response.addCookie(cookieLastName);
```

If a cookie with the same name already exists in the browser, its value is updated; otherwise, a new cookie is created.

Cookies are automatically sent to the Web server with each request from the client. The servlet retrieves all the cookies into an array using the `getCookies` method (line 100):

```
Cookie[] cookies = request.getCookies();
```

To obtain the name of the cookie, use the `getName` method (line 104):

```
String name = cookies[i].getName();
```

The cookie's value can be obtained using the `getValue` method:

```
String value = cookies[i].getValue();
```

Cookies are stored as strings just like form parameters and hidden values. If a cookie represents a numeric value, you have to convert it into an integer or a double, using the `parseInt` method in the `Integer` class or the `parseDouble` method in the `Double` class.

By default, a newly created cookie persists until the browser exits. However, you can set an expiration date, using the `setMaxAge` method, to allow a cookie to stay in the browser for up to 2,147,483,647 seconds (approximately 24,855 days).

37.8.3 Session Tracking Using the Servlet API

You have now learned both session tracking using hidden values and session tracking using cookies. These two session-tracking methods have problems. They send data to the browser either as hidden values or as cookies. The data are not secure, and anybody with knowledge of computers can obtain them. The hidden data are in HTML form, which can be viewed from the browser. Cookies are stored in the Cache directory of the browser. Because of security concerns, some browsers do not accept cookies. The client can turn the cookies off and limit their number. Another problem is that hidden data and cookies pass data as strings. You cannot pass objects using these two methods.

To address these problems, Java servlet API provides the `javax.servlet.http.HttpSession` interface, which provides a way to identify a user across more than one page request or visit to a website and to store information about that user. The servlet container uses this interface to create a session between an HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user. A session usually corresponds to one user, who may visit a site many times. The session enables tracking of a large set of data. The data can be stored as objects and are secure because they are kept on the server side.

To use the Java servlet API for session tracking, first create a session object using the `getSession()` method in the `HttpServletRequest` interface:

```
HttpSession session = request.getSession();
```

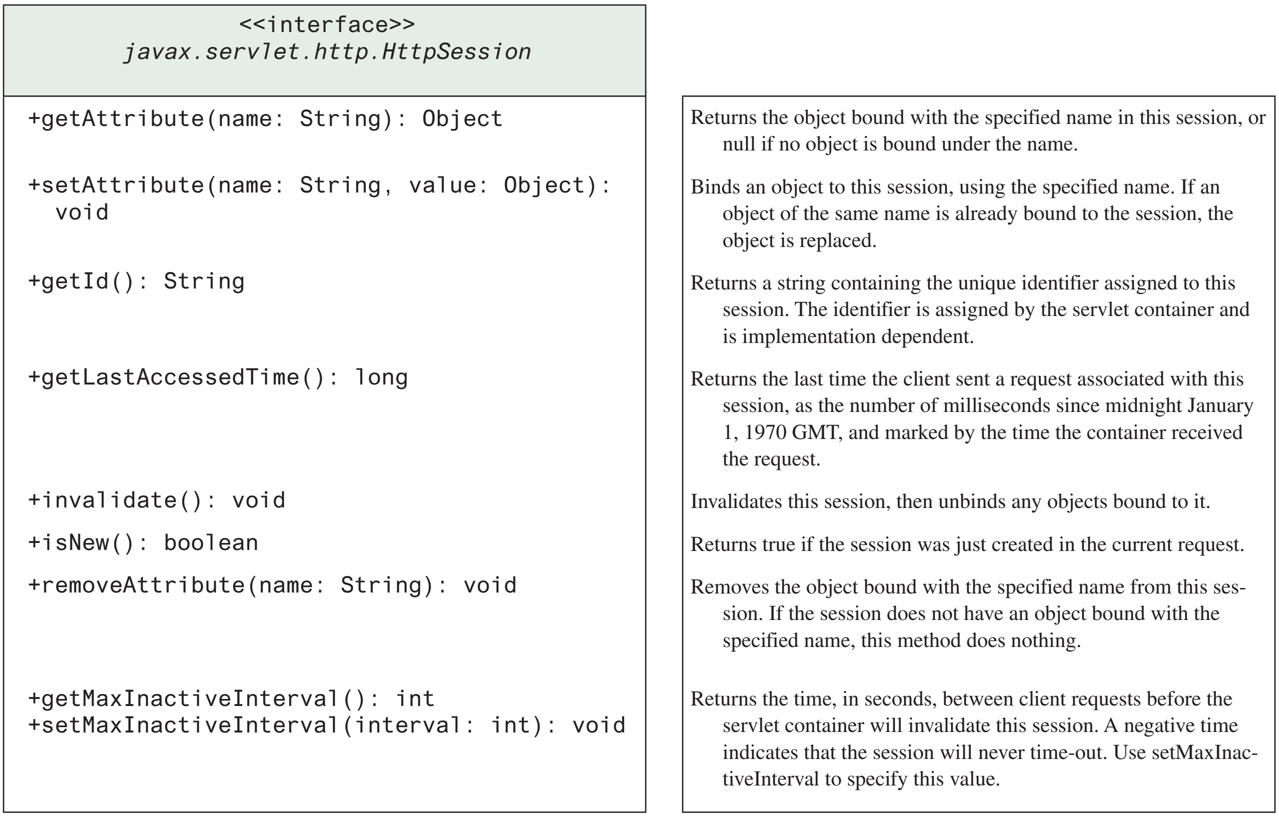
This obtains the session or creates a new session if the client does not have a session on the server.

The `HttpSession` interface provides the methods for reading and storing data to the session, and for manipulating the session, as shown in Figure 37.24.

Note

HTTP is stateless. So how does the server associate a session with multiple requests from the same client? This is handled behind the scenes by the servlet container and is transparent to the servlet programmer.

To demonstrate using `HttpSession`, let us rewrite Listing 37.9, Registration.java, and Listing 37.10, RegistrationWithCookie.java. Instead of using hidden values or cookies for session tracking, it uses servlet `HttpSession`.



+getAttribute(name: String): Object

Returns the object bound with the specified name in this session, or null if no object is bound under the name.

+setAttribute(name: String, value: Object): void

Binds an object to this session, using the specified name. If an object of the same name is already bound to the session, the object is replaced.

+getId(): String

Returns a string containing the unique identifier assigned to this session. The identifier is assigned by the servlet container and is implementation dependent.

+getLastAccessedTime(): long

Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT, and marked by the time the container received the request.

+invalidate(): void

Invalidates this session, then unbinds any objects bound to it.

+isNew(): boolean

Returns true if the session was just created in the current request.

+removeAttribute(name: String): void

Removes the object bound with the specified name from this session. If the session does not have an object bound with the specified name, this method does nothing.

+getMaxInactiveInterval(): int
+setMaxInactiveInterval(interval: int): void

Returns the time, in seconds, between client requests before the servlet container will invalidate this session. A negative time indicates that the session will never time-out. Use setMaxInactiveInterval to specify this value.

FIGURE 37.24 `HttpSession` establishes a persistent session between a client with multiple requests and the server.

Create the servlet named `RegistrationWithHttpSession` in Listing 37.11. Create an HTML file named `RegistrationWithHttpSession.html` that is identical to `Registration.html` except that the action is replaced by `RegistrationWithHttpSession`.

LISTING 37.11 RegistrationWithHttpSession.java

```java
package chapter37;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;

public class RegistrationWithHttpSession extends HttpServlet {
  // Use a prepared statement to store a student into the database
  private PreparedStatement pstmt;

  /** Initialize variables */
  public void init() throws ServletException {
    initializeJdbc();
  }

  /** Process the HTTP Get request */
  public void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
```

```
    // Set response type and output stream to the browser
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    // Obtain data from the form
    String lastName = request.getParameter("lastName");
    String firstName = request.getParameter("firstName");
    String mi = request.getParameter("mi");
    String telephone = request.getParameter("telephone");
    String email = request.getParameter("email");
    String street = request.getParameter("street");
    String city = request.getParameter("city");
    String state = request.getParameter("state");
    String zip = request.getParameter("zip");

    if (lastName.length() == 0 || firstName.length() == 0) {
      out.println("Last Name and First Name are required");
    }
    else {
      // Create an Address object
      Address address = new Address();
      address.setLastName(lastName);
      address.setFirstName(firstName);
      address.setMi(mi);
      address.setTelephone(telephone);
      address.setEmail(email);
      address.setStreet(street);
      address.setCity(city);
      address.setState(state);
      address.setZip(zip);

      // Get an HttpSession or create one if it does not exist
      HttpSession httpSession = request.getSession();

      // Store student object to the session
      httpSession.setAttribute("address", address);

      // Ask for confirmation
      out.println("You entered the following data");
      out.println("<p>Last name: " + lastName);
      out.println("<p>First name: " + firstName);
      out.println("<p>MI: " + mi);
      out.println("<p>Telephone: " + telephone);
      out.println("<p>Email: " + email);
      out.println("<p>Address: " + street);
      out.println("<p>City: " + city);
      out.println("<p>State: " + state);
      out.println("<p>Zip: " + zip);

      // Set the action for processing the answers
      out.println("<p><form method=\"post\" action=" +
        "RegistrationWithHttpSession>");
      out.println("<p><input type=\"submit\" value=\"Confirm\" >");
      out.println("</form>");
    }

    out.close(); // Close stream
  }

  /** Process the HTTP Post request */
  public void doPost(HttpServletRequest request, HttpServletResponse
      response) throws ServletException, IOException {
    // Set response type and output stream to the browser
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    // Obtain the HttpSession
    HttpSession httpSession = request.getSession();

    // Get the Address object in the HttpSession
    Address address = (Address)(httpSession.getAttribute("address"));

    try {
      storeStudent(address);

      out.println(address.getFirstName() + " " + address.getLastName()
        + " is now registered in the database");
      out.close(); // Close stream
    }
    catch(Exception ex) {
      out.println("Error: " + ex.getMessage());
    }
  }

  /** Initialize database connection */
  private void initializeJdbc() {
    try {
      // Load the JDBC driver
      Class.forName("com.mysql.jdbc.Driver");
      System.out.println("Driver loaded");

      // Establish a connection
      Connection conn = DriverManager.getConnection
        ("jdbc:mysql://localhost/javabook" , "scott", "tiger");
      System.out.println("Database connected");

      // Create a Statement
      pstmt = conn.prepareStatement("insert into Address " +
        "(lastName, firstName, mi, telephone, email, street, city, "
        + "state, zip) values (?, ?, ?, ?, ?, ?, ?, ?, ?)");
    }
    catch (Exception ex) {
      System.out.println(ex);
    }
  }

  /** Store an address to the database */
  private void storeStudent(Address address) throws SQLException {
    pstmt.setString(1, address.getLastName());
    pstmt.setString(2, address.getFirstName());
    pstmt.setString(3, address.getMi());
    pstmt.setString(4, address.getTelephone());
    pstmt.setString(5, address.getEmail());
    pstmt.setString(6, address.getStreet());
    pstmt.setString(7, address.getCity());
    pstmt.setString(8, address.getState());
    pstmt.setString(9, address.getZip());
    pstmt.executeUpdate();
  }
}
```

The statement (line 52)

```
HttpSession httpSession = request.getSession();
```

obtains a session, or creates a new session if the session does not exist.

Since objects can be stored in **HttpSession**, this program defines an **Address** class. An **Address** object is created and is stored in the session using the **setAttribute** method, which binds the object with a name like the one shown below (line 55):

```
httpSession.setAttribute("address", address);
```

To retrieve the object, use the following statement (line 90):

```
Address address = (Address)(httpSession.getAttribute("address"));
```

There is only one session between a client and a servlet. You can store any number of objects in a session. By default, the maximum inactive interval on many Web servers including Tomcat and GlassFish is 1800 seconds (i.e., a half-hour), meaning that the session expires if there is no activity for 30 minutes. You can change the default using the **setMaxInactiveInterval** method. For example, to set the maximum inactive interval to one hour, use

```
httpSession.setMaxInactiveInterval(3600);
```

If you set a negative value, the session will never expire.

For this servlet program to work, you have to create the **Address** class in NetBeans, as follows:

1. Choose *New*, *Java Class* from the context menu of the **liangweb** node in the project pane to display the New Java Class dialog box.

2. Enter **Address** as the Class Name and **chapter37** as the package name. Click *Finish* to create the class.

3. Enter the code, as shown in Listing 37.12.

Listing 37.12 Address.java

```
package chapter37;

public class Address {
private String firstName;
private String mi;
private String lastName;
private String telephone;
private String street;
private String city;
private String state;
private String email;
private String zip;

  public String getFirstName() {
    return this.firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getMi() {
    return this.mi;
  }

  public void setMi(String mi) {
    this.mi = mi;
  }

  public String getLastName() {
    return this.lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public String getTelephone() {
    return this.telephone;
  }

  public void setTelephone(String telephone) {
    this.telephone = telephone;
  }

  public String getEmail() {
    return this.email;
  }

  public void setEmail(String email) {
    this.email = email;
  }

  public String getStreet() {
    return this.street;
  }

  public void setStreet(String street) {
    this.street = street;
  }

  public String getCity() {
    return this.city;
  }

  public void setCity(String city) {
    this.city = city;
  }

  public String getState() {
    return this.state;
  }

  public void setState(String state) {
    this.state = state;
  }

  public String getZip() {
    return this.zip;
  }

  public void setZip(String zip) {
    this.zip = zip;
  }
}
```

This support class will also be reused in the upcoming chapters.

Check Point

37.8.1 What is session tracking? What are three techniques for session tracking?

37.8.2 How do you create a cookie, send a cookie to a browser, get cookies from a browser, get the name of a cookie, set a new value in the cookie, and set cookie expiration time?

37.8.3 Do you have to create five `Cookie` objects in the servlet in order to send five cookies to the browser?

37.8.4 How do you get a session, set object value for the session, and get object value from the session?

37.8.5 Suppose you inserted the following code in line 53 in Listing 37.11:

```
httpSession.setMaxInactiveInterval(1);
```

What would happen after the user clicked the *Confirm* button from the browser? Test your answer by running the program.

37.8.6 Suppose you inserted the following code in line 53 in Listing 37.11:

```
httpSession.setMaxInactiveInterval(-1);
```

What would happen after the user clicked the *Confirm* button from the browser?

Key Terms

Common Gateway Interface 37-3	URL query string 37-3
CGI programs 37-3	servlet 37-2
cookie 37-30	servlet container (servlet engine) 37-4
GET and POST methods 37-3	servelt life-cycle methods 37-10
GlassFish 37-5	Tomcat 37-5
HTML form 37-15	

Chapter Summary

1. A servlet is a special kind of program that runs from a Web server. Tomcat and GlassFish are Web servers that can run servlets.

2. A servlet URL is specified by the host name, port, and request string (e.g., http://localhost:8084/liangweb/ServletClass). There are several ways to invoke a servlet: (1) by typing a servlet URL from a Web browser, (2) by placing a hyper link in an HTML page, and (3) by embedding a servlet URL in an HTML form. All the requests trigger the GET method, except that in the HTML form you can explicitly specify the POST method.

3. You develop a servlet by defining a class that extends the `HttpServlet` class, implements the `doGet(HttpServletRequest, HttpServletResponse)` method to respond to the GET method, and implements the `doPost(HttpServletRequest, HttpServletResponse)` method to respond to the POST method.

4. The request information passed from a client to the servlet is contained in an object of `HttpServletRequest`. You can use the methods `getParameter`, `getParameterValues`, `getRemoteAddr`, `getRemoteHost`, `getHeader`, `getQueryString`, `getCookies`, and `getSession` to obtain the information from the request.

5. The content sent back to the client is contained in an object of `HttpServletResponse`. To send content to the client, first set the type of the content (e.g., html/plain) using the `setContentType(contentType)` method, then output the content through an I/O stream on the `HttpServletResponse` object. You can obtain a character `PrintWriter` stream using the `getWriter()` method and obtain a binary `OutputStream` using the `getOutputStream()` method.

6. A servlet may be shared by many clients. When the servlet is first created, its `init` method is called. It is not called again as long as the servlet is not destroyed. The `service` method is invoked each time the server receives a request for the servlet. The server spawns a new thread and invokes `service`. The `destroy` method is invoked after a timeout period has passed or the Web server is stopped.

7. There are three ways to track a session. You can track a session by passing data from the servlet to the client as a hidden value in a dynamically generated HTML form by including a field such as `<input type="hidden" name="lastName" value="Smith">`. The next request will submit the data back to the servlet. The servlet retrieves this hidden value just like any other parameter value using the `getParameter` method.

8. You can track sessions using cookies. A cookie is created using the constructor `new Cookie(String name, String value)`. Cookies are sent from the server through the object of `HttpServletResponse` using the `addCookie(aCookie)` method to tell the browser to add a cookie with a given name and its associated value. If the browser already has a cookie with the key name, the value will be updated. The browser will then send the cookie with any request submitted to the same server. Cookies can have expiration dates set, after which they will not be sent to the server.

9. Java servlet API provides a session-tracking tool that enables tracking of a large set of data. A session can be obtained using the `getSession()` method through an `HttpServletRequest` object. The data can be stored as objects and are secure because they are kept on the server side using the `setAttribute(String name, Object value)` method.

Quiz

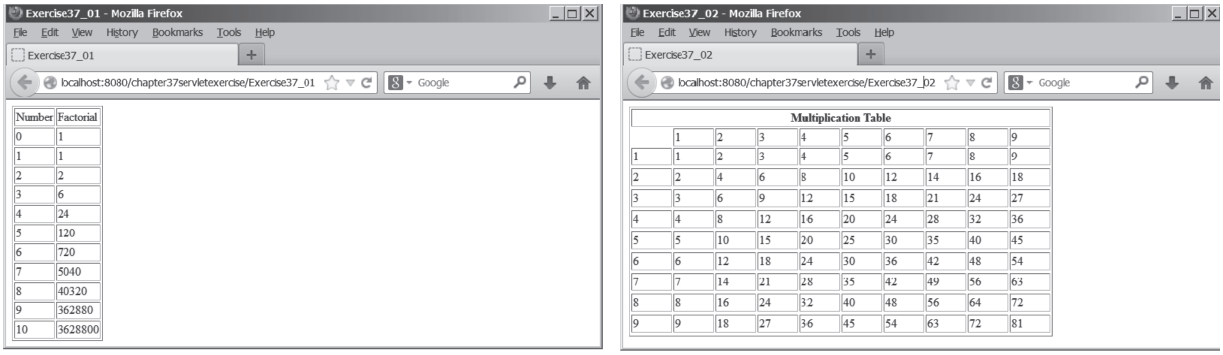
Answer the quiz for this chapter online at the book Companion Website.

Programming Exercises

MyProgrammingLab™

Section 37.5

***37.1** (*Factorial table*) Write a servlet to display a table that contains factorials for the numbers from `0` to `10`, as shown in Figure 37.25.


Exercise37_01 - Mozilla Firefox
localhost:8080/chapter37servletexercise/Exercise37_01
Exercise37_02 - Mozilla Firefox
localhost:8080/chapter37servletexercise/Exercise37_02

Number	Factorial
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800

(a)

Multiplication Table

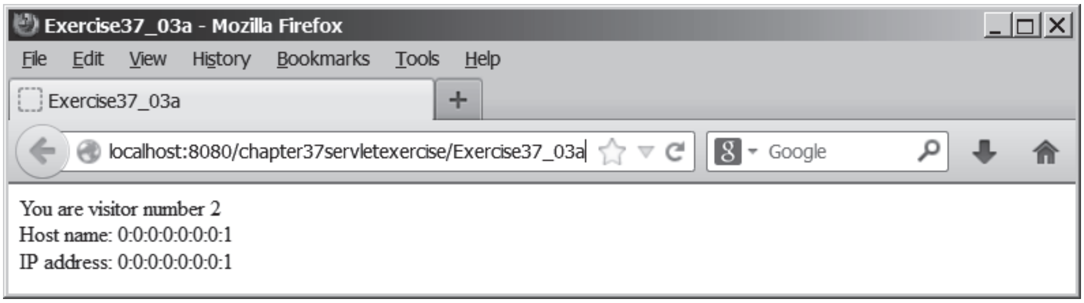
	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

(b)

FIGURE 37.25 (a) The servlet displays factorials for the numbers from 0 to 10 in a table. (b) The servlet displays the multiplication table.

***37.2** (*Multiplication table*) Write a servlet to display a multiplication table, as shown in Figure 37.25b.

***37.3** (*Visit count*) Develop a servlet that displays the number of visits on the servlet. Also display the client's host name and IP address, as shown in Figure 37.26.


Exercise37_03a - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Exercise37_03a
localhost:8080/chapter37servletexercise/Exercise37_03a
Google

You are visitor number 2
Host name: 0:0:0:0:0:0:0:1
IP address: 0:0:0:0:0:0:0:1

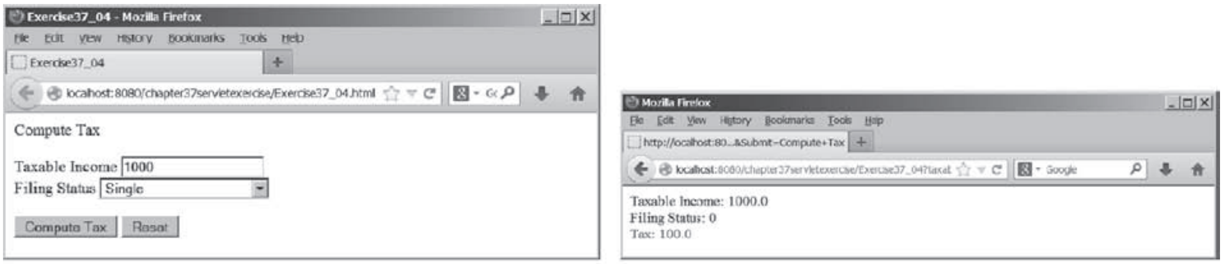
FIGURE 37.26 The servlet displays the number of visits and the client's host name, IP address, and request URL.

Implement this program in three different ways:

1. Use an instance variable to store `count`. When the servlet is created for the first time, `count` is `0`. `count` is incremented every time the servlet's `doGet` method is invoked. When the Web server stops, `count` is lost.
2. Store the count in a file named **Exercise39_3.dat**, and use `RandomAccessFile` to read the count in the servlet's `init` method. The count is incremented every time the servlet's `doGet` method is invoked. When the Web server stops, store the count back to the file.
3. Instead of counting total visits from all clients, count the visits by each client identified by the client's IP address. Use `Map` to store a pair of IP addresses and visit counts. For the first visit, an entry is created in the map. For subsequent visits, the visit count is updated.

Section 37.6

***37.4** (*Calculate tax*) Write an HTML form to prompt the user to enter taxable income and filing status, as shown in Figure 37.27a. Clicking the *Compute Tax* button invokes a servlet to compute and display the tax, as shown in Figure 37.27b. Use the `computeTax` method introduced in Listing 3.7, ComputingTax.java, to compute tax.


Exercise37_04 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
localhost:8080/chapter37servletexercise/Exercise37_04.html
Compute Tax
Reset
Mozilla Firefox
http://localhost:80...&Submit=Compute+Tax
localhost:8080/chapter37servletexercise/Exercise37_04?taxat

Compute Tax

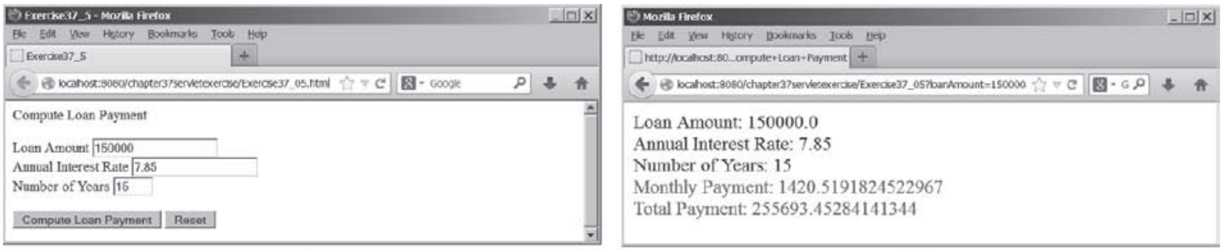
Taxable Income 1000
Filing Status Single

```
Taxable Income: 1000.0
Filing Status: 0
Tax: 100.0
```

(a) (b)

FIGURE 37.27 The servlet computes the tax.

***37.5** (*Calculate loan*) Write an HTML form that prompts the user to enter loan amount, interest rate, and number of years, as shown in Figure 37.28a. Clicking the *Compute Loan Payment* button invokes a servlet to compute and display the monthly and total loan payments, as shown in Figure 37.28b. Use the Loan class given in Listing 10.2, Loan.java, to compute the monthly and total payments.


Exercise37_5 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
localhost:8080/chapter37servletexercise/Exercise37_05.html
Compute Loan Payment
Reset
Mozilla Firefox
http://localhost:80...ompute+Loan+Payment
localhost:8080/chapter37servletexercise/Exercise37_05?loanAmount=150000

Compute Loan Payment

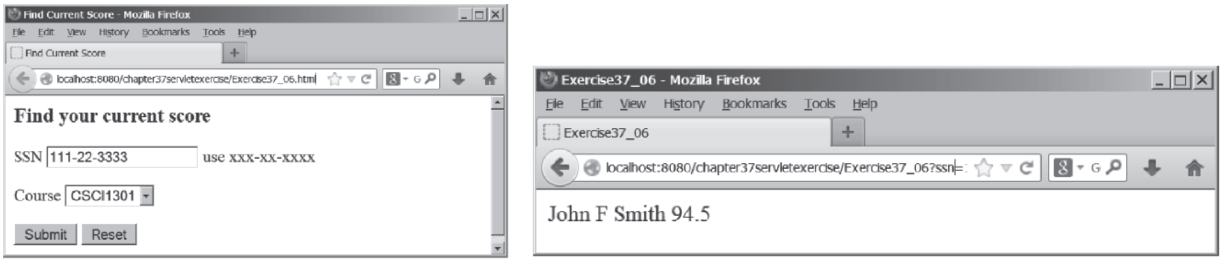
Loan Amount 150000
Annual Interest Rate 7.85
Number of Years 15

```
Loan Amount: 150000.0
Annual Interest Rate: 7.85
Number of Years: 15
Monthly Payment: 1420.5191824522967
Total Payment: 255693.45284141344
```

(a) (b)

FIGURE 37.28 The servlet computes the loan payment.

****37.6** (*Find scores from text files*) Write a servlet that displays the student name and the current score, given the SSN and class ID. For each class, a text file is used to store the student name, SSN, and current score. The file is named after the class ID with .txt extension. For instance, if the class ID were csci1301, the file name would be csci1301.txt. Suppose each line consists of student name, SSN, and score. These three items are separated by the # sign. Create an HTML form that enables the user to enter the SSN and class ID, as shown in Figure 37.29a. Upon clicking the *Submit* button, the result is displayed, as shown in Figure 37.29b. If the SSN or the class ID does not match, report an error. Assume three courses are available: CSCI1301, CSCI1302, and CSCI3720.


Find Current Score - Mozilla Firefox
File Edit View History Bookmarks Tools Help
localhost:8080/chapter37servletexercise/Exercise37_06.html
Submit
Reset
Exercise37_06 - Mozilla Firefox
localhost:8080/chapter37servletexercise/Exercise37_06?ssn=

Find your current score

SSN 111-22-3333 use xxx-xx-xxxx

Course CSCI1301

```
John F Smith 94.5
```

(a) (b)

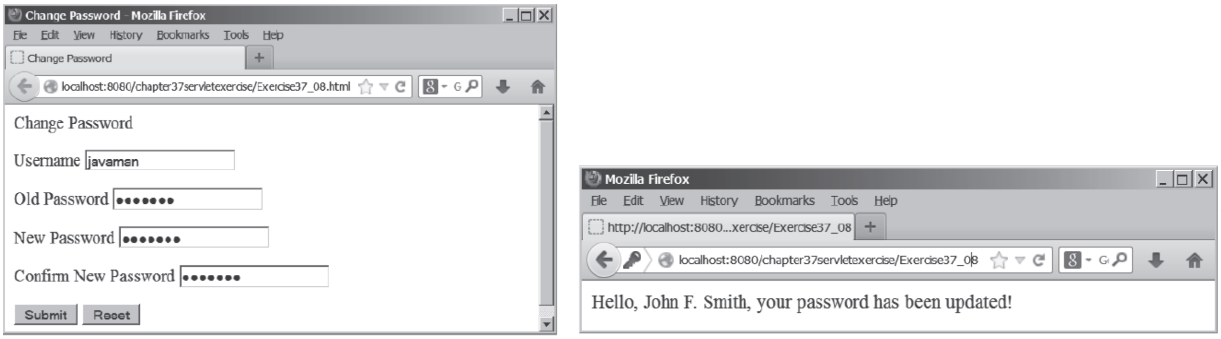
FIGURE 37.29 The HTML form accepts the SSN and class ID from the user and sends them to the servlet to obtain the score.

Section 37.7

****37.7** (*Find scores from database tables*) Rewrite the preceding servlet. Assume for each class, a table is used to store the student name, ssn, and score. The table name is the same as the class ID. For instance, if the class ID were csci1301, the table name would be csci1301.

***37.8** (*Change the password*) Write a servlet that enables the user to change the password from an HTML form, as shown in Figure 37.30a. Suppose the user information is stored in a database table named Account with three columns: username, password, and name, where name is the real name of the user. The servlet performs the following tasks:

a. Verify that the username and old password are in the table. If not, report the error and redisplay the HTML form.
b. Verify that the new password and the confirmed password are the same. If not, report this error and redisplay the HTML form.
c. If the user information is entered correctly, update the password and report the status of the update to the user, as shown in Figure 37.30b.


Change Password - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Change Password
localhost:8080/chapter37servletexercise/Exercise37_08.html
Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080...xercise/Exercise37_08
localhost:8080/chapter37servletexercise/Exercise37_08

Change Password

Username javaman

Old Password •••••••

New Password •••••••

Confirm New Password •••••••

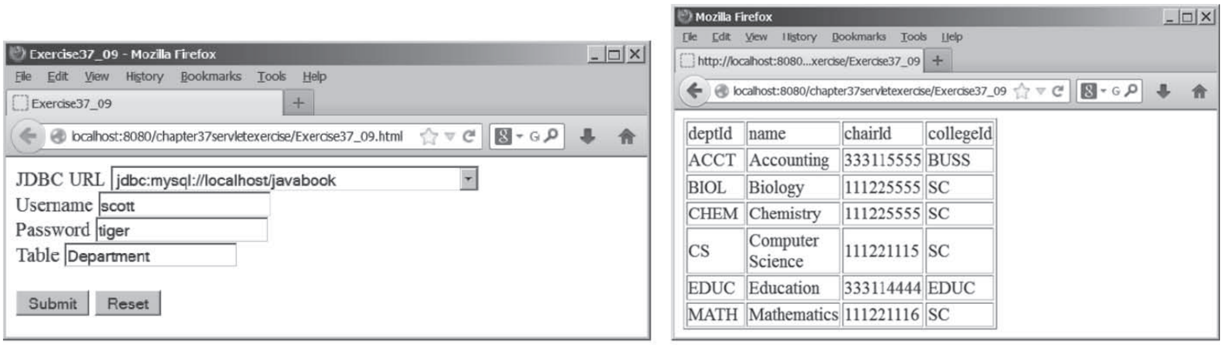
Submit Reset

Hello, John F. Smith, your password has been updated!

(a) (b)

FIGURE 37.30 The user enters the username and the old password and sets a new password. The servlet reports the status of the update to the user.

****37.9** (*Display database tables*) Write an HTML form that prompts the user to enter or select a JDBC driver, database URL, username, password, and table name, as shown in Figure 37.31a. Clicking the *Submit* button displays the table content, as shown in Figure 37.31b.


Exercise37_09 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Exercise37_09
localhost:8080/chapter37servletexercise/Exercise37_09.html
Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080...xercise/Exercise37_09
localhost:8080/chapter37servletexercise/Exercise37_09

JDBC URL jdbc:mysql://localhost/javabook

Username scott

Password tiger

Table Department

Submit Reset

deptId	name	chairId	collegeId
ACCT	Accounting	333115555	BUSS
BIOL	Biology	111225555	SC
CHEM	Chemistry	111225555	SC
CS	Computer Science	111221115	SC
EDUC	Education	333114444	EDUC
MATH	Mathematics	111221116	SC

(a) (b)

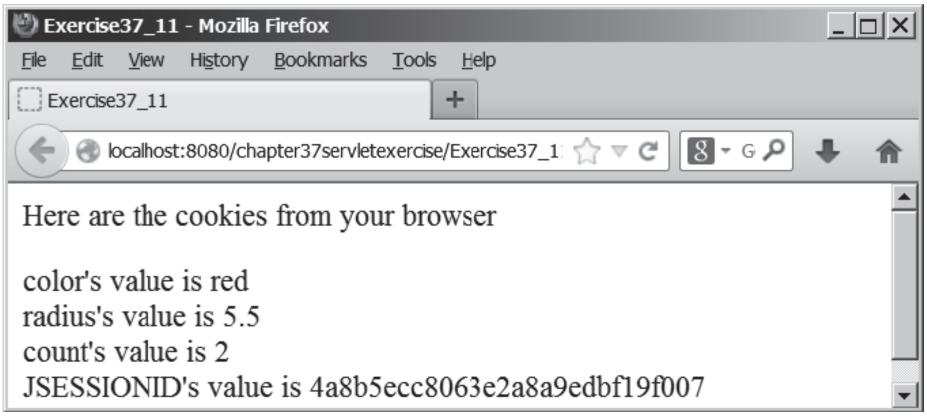
FIGURE 37.31 The user enters database information and specifies a table to display its content.

Section 37.8

***37.10** (*Store cookies*) Write a servlet that stores the following cookies in a browser, and set their max age for two days.

Cookie 1: name is "color" and value is red.
Cookie 2: name is "radius" and value is 5.5.
Cookie 3: name is "count" and value is 2.

***37.11** (*Retrieve cookies*) Write a servlet that displays all the cookies on the client. The client types the URL of the servlet from the browser to display all the cookies stored on the browser. (see Figure 37.32.)

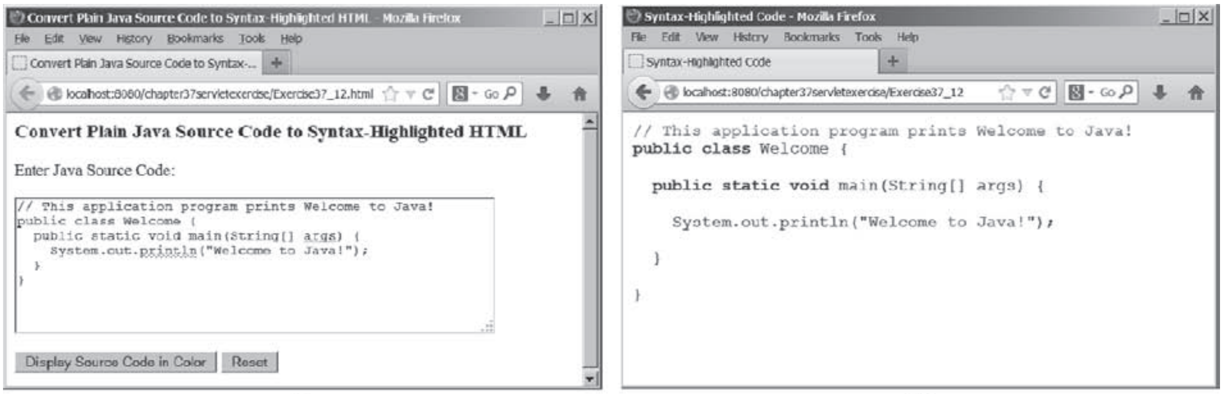

Exercise37_11 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Exercise37_11
localhost:8080/chapter37servletexercise/Exercise37_1

```
Here are the cookies from your browser

color's value is red
radius's value is 5.5
count's value is 2
JSESSIONID's value is 4a8b5ecc8063e2a8a9edbf19f007
```

FIGURE 37.32 All the cookies on the client are displayed in the browser.

Comprehensive

*****37.12** (*Syntax highlighting*) Create an HTML form that prompts the user to enter a Java program in a text area, as shown in Figure 37.33a. The form invokes a servlet that displays the Java source code in a syntax-highlighted HTML format, as shown in Figure 37.33b. The keywords, comments, and literals are displayed in bold navy, green, and blue, respectively.


Convert Plain Java Source Code to Syntax-Highlighted HTML - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Convert Plain Java Source Code to Syntax-...
localhost:8080/chapter37servletexercise/Exercise37_12.html
Display Source Code in Color
Reset
Syntax-Highlighted Code - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Syntax-Highlighted Code
localhost:8080/chapter37servletexercisa/Exercise37_12

Convert Plain Java Source Code to Syntax-Highlighted HTML

Enter Java Source Code:

```
// This application program prints Welcome to Java!
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

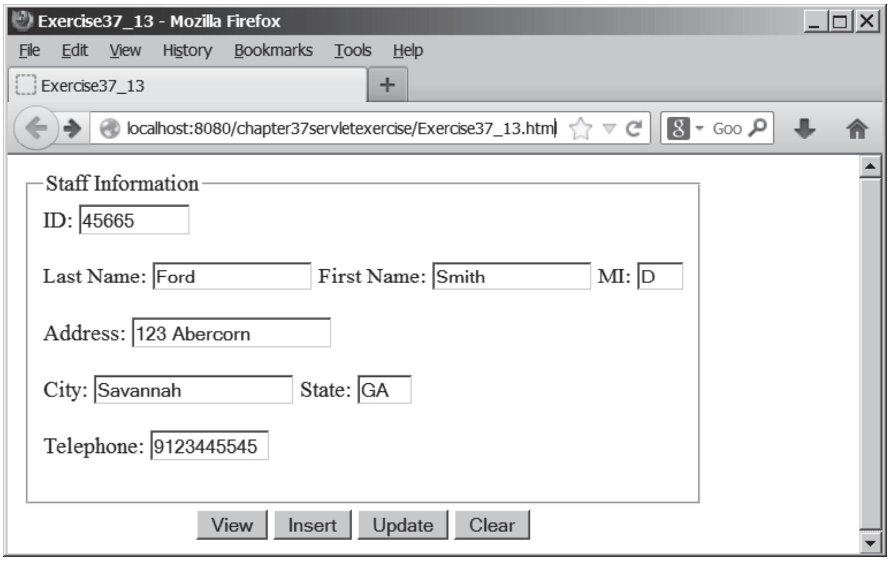
(a)

```
// This application program prints Welcome to Java!
public class Welcome {

  public static void main(String[] args) {

    System.out.println("Welcome to Java!");

  }

}
```

(b)

FIGURE 37.33 The Java code in plain text in (a) is displayed in HTML with syntax highlighted in (b).

****37.13** (*Access and update a* `Staff` *table*) Write a Java servlet for Exercise 33.1, as shown in Figure 37.34.


Exercise37_13 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Exercise37_13
localhost:8080/chapter37servletexercise/Exercise37_13.html
View
Insert
Update
Clear

Staff Information

ID: 45665

Last Name: Ford First Name: Smith MI: D

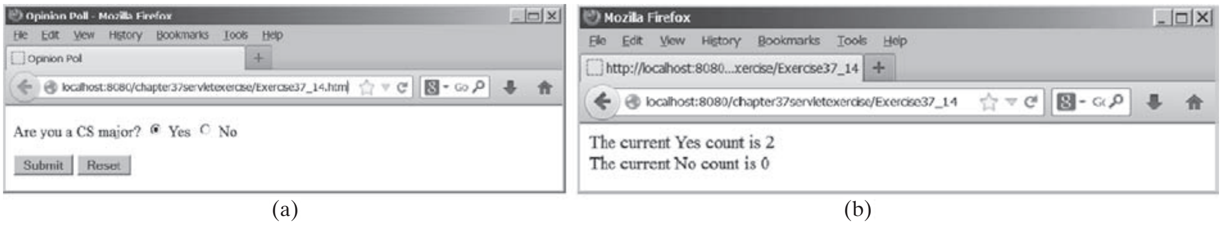
Address: 123 Abercorn

City: Savannah State: GA

Telephone: 9123445545

FIGURE 37.34 The webpage lets you view, insert, and update staff information.

*****37.14** (*Opinion poll*) Create an HTML form that prompts the user to answer a question such as "Are you a CS major?", as shown in Figure 37.35a. When the *Submit* button is clicked, the servlet increases the Yes or No count in a database and displays the current Yes and No counts, as shown in Figure 37.35b.


Opinion Poll - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Opinion Poll
localhost:8080/chapter37servletexercise/Exercise37_14.html
Submit
Reset
Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080...xercise/Exercise37_14
localhost:8080/chapter37servletexercise/Exercise37_14

Are you a CS major? ● Yes ○ No

The current Yes count is 2
The current No count is 0

(a) (b)

FIGURE 37.35 The HTML form prompts the user to enter Yes or No for a question in (a), and the servlet updates the Yes or No counts (b).

Create a table named `Poll`, as follows:

```
create table Poll (
  question varchar(40) primary key,
  yesCount int,
  noCount int);
```

Insert one row into the table, as follows:

```
insert into Poll values ('Are you a CS major? ', 0, 0);
```

CHAPTER

38

Javaserver Pages

Objectives

- To create a simple JSP page (§38.2).
- To explain how a JSP page is processed (§38.3).
- To use JSP constructs to code JSP script (§38.4).
- To use predefined variables and directives in JSP (§§38.5–38.6).
- To use JavaBeans components in JSP (§38.7).
- To get and set JavaBeans properties in JSP (§38.8).
- To associate JavaBeans properties with input parameters (§38.9).
- To forward requests from one JSP page to another (§38.10).
- To develop an application for browsing database tables using JSP (§38.11).

38.1 Introduction

JavaServer Pages are the Java scripts and code embedded in an HTML file.

Servlets can be used to generate dynamic Web content. One drawback, however, is that you have to embed HTML tags and text inside the Java source code. Using servlets, you have to modify the Java source code and recompile it if changes are made to the HTML text. If you have a lot of HTML script in a servlet, the program is difficult to read and maintain, since the HTML text is part of the Java program. JavaServer Pages (JSP) was introduced to remedy this drawback. JSP enables you to write regular HTML script in the normal way and embed Java code to produce dynamic content.

38.2 Creating a Simple JSP Page

An IDE such an NetBeans is an effecitve tools for creating JavaServer Pages.

JSP provides an easy way to create dynamic webpages and simplify the task of building Web applications. A JavaServer page is like a regular HTML page with special tags, known as *JSP tags*, which enable the Web server to generate dynamic content. You can create a webpage with HTML script and enclose the Java code for generating dynamic content in the JSP tags. Here is an example of a simple JSP page:

```
<!-- CurrentTime.jsp -->
<html>
  <head>
    <title>
      CurrentTime
    </title>
  </head>
  <body>
    Current time is <%= new java.util.Date() %>
  </body>
</html>
```

The dynamic content is enclosed in the tag that begins with `<%=` and ends with `%>`. The current time is returned as a string by invoking the `toString` method of an object of the `java.util.Date` class.

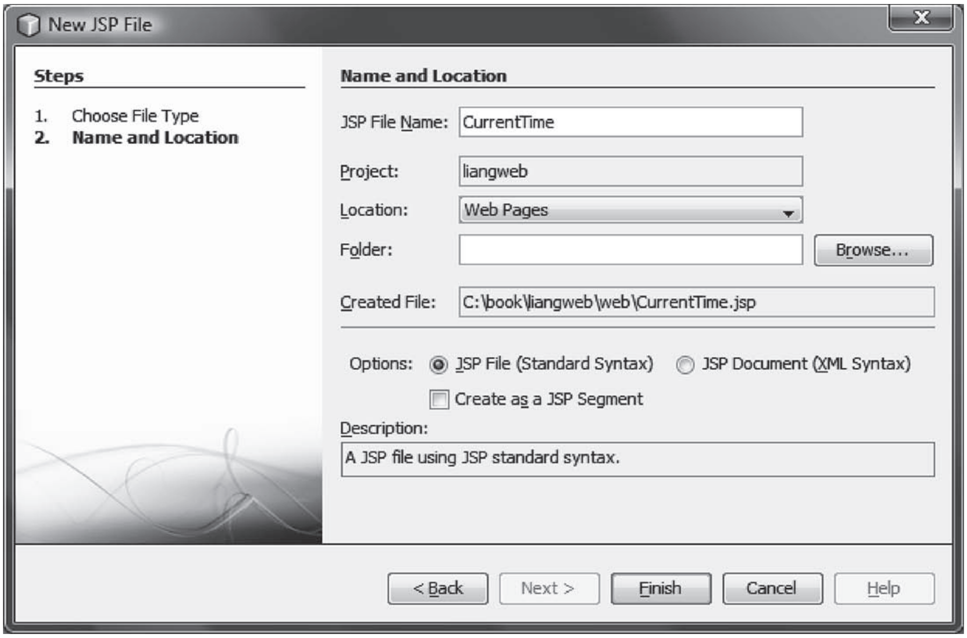
An IDE like NetBeans can greatly simplify the task of developing JSP. To create JSP in NetBeans, first you need to create a Web project. A Web project named `liangweb` was created in the preceding chapter. For convenience, this chapter will create JSP in the `liangweb` project.

Here are the steps to create and run CurrentTime.jsp:

1. Right-click the `liangweb` node in the project pane and choose **New, JSP** to display the New JSP dialog box, as shown in Figure 38.1.
2. Enter `CurrentTime` in the JSP File Name field and click *Finish*. You will see CurrentTime.jsp appearing under the webpages node in `liangweb`.
3. Complete the code for CurrentTime.jsp, as shown in Figure 38.2.
4. Right-click CurrentTime.jsp in the project pane and choose *Run File*. You will see the JSP page displayed in a Web browser, as shown in Figure 38.3.

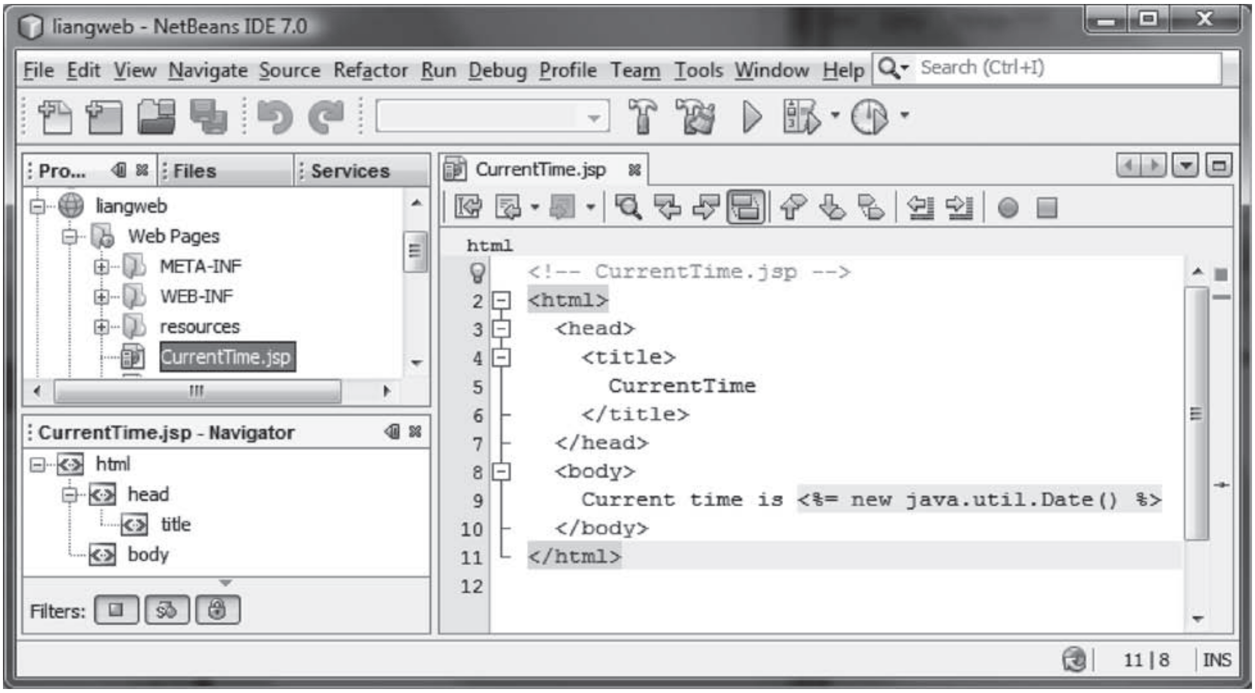
Note

Like servlets, you can develop JSP in NetBeans, create a .war file, and then deploy the .war file in a Java Web server such as Tomcat and GlassFish.



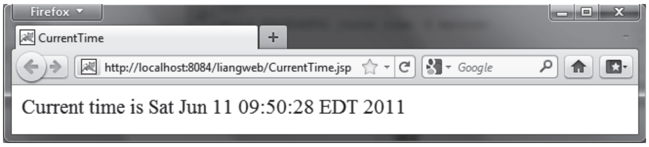
JSP File Name: CurrentTime
Project: liangweb
Location: Web Pages
Folder:
Created File: C:\book\liangweb\web\CurrentTime.jsp
Options: ● JSP File (Standard Syntax) ○ JSP Document (XML Syntax)
□ Create as a JSP Segment
Description:
A JSP file using JSP standard syntax.

FIGURE 38.1 You can create a JSP page using NetBeans.



```
<!-- CurrentTime.jsp -->
<html>
  <head>
    <title>
      CurrentTime
    </title>
  </head>
  <body>
    Current time is <%= new java.util.Date() %>
  </body>
</html>
```

FIGURE 38.2 A template for a JSP page is created.



```
Current time is Sat Jun 11 09:50:28 EDT 2011
```

FIGURE 38.3 The result from a JSP page is displayed in a Web browser.

38.3 How Is a JSP Page Processed?

Key Point

JavaServer Pages are preprocessed and compiled into Java servlets by a Java Web server.

A JSP page must first be processed by a Web server before it can be displayed in a Web browser. The Web server must support JSP, and the JSP page must be stored in a file with a .jsp extension. The Web server translates the .jsp file into a Java servlet, compiles the servlet, and executes it. The result of the execution is sent to the browser for display. Figure 38.4 shows how a JSP page is processed by a Web server.

Note

A JSP page is translated into a servlet when the page is requested for the first time. It is not retranslated if the page is not modified. To ensure that the first-time real user does not encounter a delay, JSP developers should test the page after it is installed.

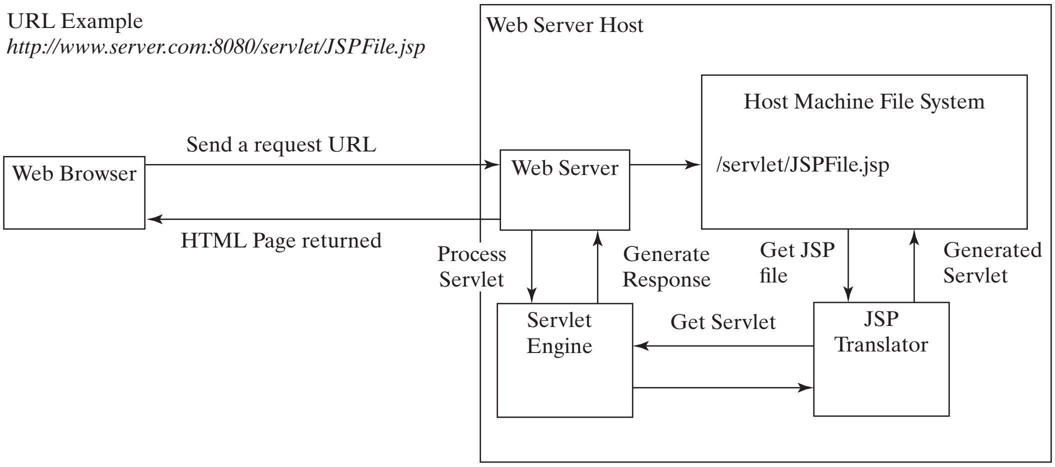


FIGURE 38.4 A JSP page is translated into a servlet.

Check Point

38.2.1 What is the file-name extension of a JavaServer page? How is a JSP page processed?

38.2.2 Can you create a .war that contains JSP in NetBeans? Where should the .war be placed in a Java application server?

38.2.3 You can display an HTML file (e.g., c:\test.html) by typing the complete file name in the Address field of Internet Explorer. Why can't you display a JSP file by simply typing the file name?

38.4 JSP Scripting Constructs

Key Point

There are three main types of JSP constructs: scripting constructs, directives, and actions.

Scripting elements enable you to specify Java code that will become part of the resultant servlet. *Directives* enable you to control the overall structure of the resultant servlet. *Actions* enable you to control the behavior of the JSP engine. This section introduces scripting constructs.

Three types of JSP scripting constructs can be used to insert Java code into a resultant servlet: expressions, scriptlets, and declarations.

A JSP *expression* is used to insert a Java expression directly into the output. It has the following form:

```
<%= Java expression %>
```

The expression is evaluated, converted into a string, and sent to the output stream of the servlet.

A JSP *scriptlet* enables you to insert a Java statement into the servlet's `jspService` method, which is invoked by the `service` method. A JSP scriptlet has the following form:

```
<% Java statement %>
```

A JSP *declaration* is for declaring methods or fields into the servlet. It has the following form:

```
<%! Java declaration %>
```

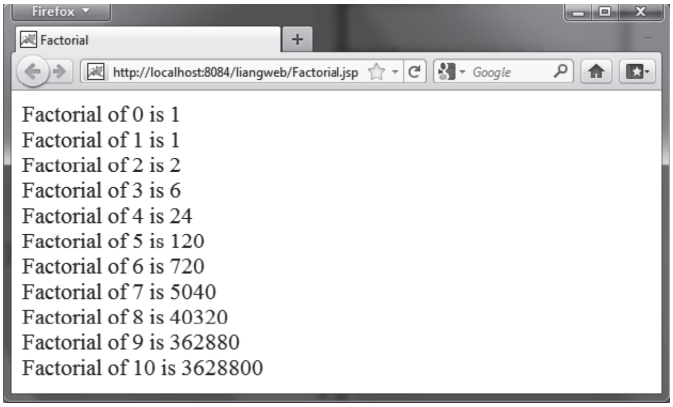
HTML comments have the following form:

```
<!-- HTML Comment -->
```

If you don't want the comment to appear in the resultant HTML file, use the following comment in JSP:

```
<%-- JSP Comment --%>
```

Listing 38.1 creates a JavaServer page that displays factorials for numbers from `0` to `10`, as shown in Figure 38.5.



Factorial of 0 is 1
Factorial of 1 is 1
Factorial of 2 is 2
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
Factorial of 6 is 720
Factorial of 7 is 5040
Factorial of 8 is 40320
Factorial of 9 is 362880
Factorial of 10 is 3628800

FIGURE 38.5 The JSP page displays factorials.

LISTING 38.1 Factorial.jsp

```
<html>
  <head>
    <title>
      Factorial
    </title>
  </head>
  <body>

  <% for (int i = 0; i <= 10; i++) { %>
      Factorial of <%= i %> is
        <%= computeFactorial(i) %> <br />
  <% } %>

  <%! private long computeFactorial(int n) {
      if (n == 0)
        return 1;
      else
        return n * computeFactorial(n - 1);
19        }
20    %>
21
22    </body>
23  </html>
```

JSP scriptlets are enclosed between `<%` and `%>`. Thus,

```
for (int i = 0; i <= 10; i++) {, (line 9)
```

is a scriptlet and as such is inserted directly into the servlet's `jspService` method.

JSP expressions are enclosed between `<%=` and `%>`. Thus,

```
<%= i %>, (line 10)
```

is an expression and is inserted into the output stream of the servlet.

JSP declarations are enclosed between `<%!` and `%>`. Thus,

```
<%! private long computeFactorial(int n) {
        ...
      }
 %>
```

is a declaration that defines methods or fields in the servlet.

What will be different if line 9 is replaced by the two alternatives shown below? Both work fine, but there is an important difference. In (a), `i` is a local variable in the servlet, whereas in (b), `i` is an instance variable when translated to the servlet.

```
<% int i = 0; %>
<% for ( ; i <= 10; i++) { %>
```

(a)

```
<%! int i; %>
<% for (i = 0; i <= 10; i++) { %>
```

(b)

Caution:
For JSP, the loop body, even though it contains a single statement, must be placed inside braces. It would be wrong to delete the opening brace (`{`) in line 9 and the closing brace (`<% } %>`) in line 12.

Caution:
There is no semicolon at the end of a JSP expression. For example, `<%= i; %>` is incorrect. But there must be a semicolon for each Java statement in a JSP scriptlet. For example, `<% int i = 0 %>` is incorrect.

Caution:
JSP and Java elements are case sensitive, but HTML is not.

Check Point

38.4.1 What are a JSP expression, a JSP scriptlet, and a JSP declaration? How do you write these constructs in JSP?

38.4.2 Find three syntax errors in the following JSP code:

```
<%! int k %>
<% for (int j = 1; j <= 9; j++) %>
   <%= j; %> <br />
```

38.4.3 In the following JSP, which variables are instance variables, and which are local variables when it is translated into in the servlet?

```
<%! int k; %>
<%! int i; %>
```

```
<% for (int j = 1; j <= 9; j++) k += 1;%>
<%= k><br /> <%= i><br /> <%= getTime()><br />
<% private long getTime() {
    long time = System.currentTimeMillis();
    return time;
  } %>
```

38.5 Predefined Variables

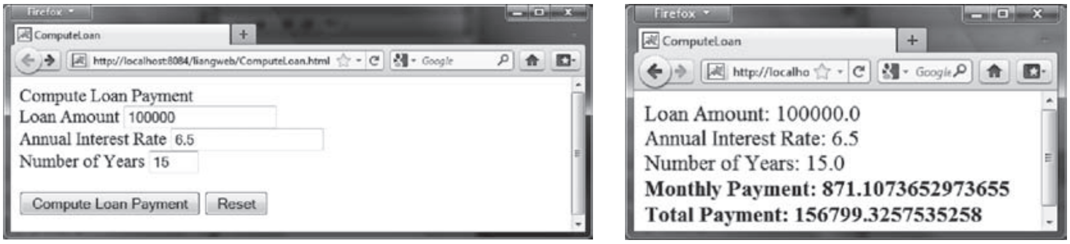
JSP provides predefined variables that can be convinently used in the JSP code.

You can use variables in JSP. For convenience, JSP provides eight predefined variables from the servlet environment that can be used with JSP expressions and scriptlets. These variables are also known as *JSP implicit objects*.

- `request` represents the client's request, which is an instance of `HttpServletRequest`. You can use it to access request parameters and HTTP headers, such as cookies and host name.
- `response` represents the servlet's response, which is an instance of `HttpServletResponse`. You can use it to set response type and send output to the client.
- `out` represents the character output stream, which is an instance of `PrintWriter` obtained from `response.getWriter()`. You can use it to send character content to the client.
- `session` represents the `HttpSession` object associated with the request, obtained from `request.getSession()`.
- `application` represents the `ServletContext` object for storing persistent data for all clients. The difference between `session` and `application` is that session is tied to one client, but `application` is for all clients to share persistent data.
- `config` represents the `ServletConfig` object for the page.
- `pageContext` represents the `PageContext` object. `PageContext` is a new class introduced in JSP to give a central point of access to many page attributes.
- `page` is an alternative to `this`.

As an example, let us write an HTML page that prompts the user to enter loan amount, annual interest rate, and number of years, as shown in Figure 38.6a. Clicking the *Compute Loan Payment* button invokes a JSP to compute and display the monthly and total loan payments, as shown in Figure 38.6b.

The HTML file is named `ComputeLoan.html` (Listing 38.2). The JSP file is named `ComputeLoan.jsp` (Listing 38.3).



Compute Loan Payment
Loan Amount 100000
Annual Interest Rate 6.5
Number of Years 15

(a)

Loan Amount: 100000.0
Annual Interest Rate: 6.5
Number of Years: 15.0
Monthly Payment: 871.1073652973655
Total Payment: 156799.3257535258

(b)

Figure 38.6 The JSP computes the loan payments.

Listing 38.2 ComputeLoan.html

```
<!-- ComputeLoan.html -->
<html>
  <head>
    <title>ComputeLoan</title>
  </head>
  <body>
    <form method = "get" action = "ComputeLoan.jsp">
    Compute Loan Payment<br />
    Loan Amount
    <input type = "text" name = "loanAmount" /><br />
    Annual Interest Rate
    <input type = "text" name = "annualInterestRate" /><br />
    Number of Years
    <input type = "text" name = "numberOfYears" size = "3" /><br />
    <p><input type = "submit" name = "Submit"
        value = "Compute Loan Payment" />
    <input type = "reset" value = "Reset" /></p>
    </form>
  </body>
</html>
```

Listing 38.3 ComputeLoan.jsp

```
 1  <!-- ComputeLoan.jsp -->
 2  <html>
 3    <head>
 4      <title>ComputeLoan</title>
 5    </head>
 6    <body>
 7    <% double loanAmount = Double.parseDouble(
 8         request.getParameter("loanAmount"));
 9       double annualInterestRate = Double.parseDouble(
10         request.getParameter("annualInterestRate"));
11       double numberOfYears = Integer.parseInt(
12         request.getParameter("numberOfYears"));
13       double monthlyInterestRate = annualInterestRate / 1200;
14       double monthlyPayment = loanAmount * monthlyInterestRate /
15        (1 – 1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12));
16       double totalPayment = monthlyPayment * numberOfYears * 12; %>
17    Loan Amount: <%= loanAmount %><br />
18    Annual Interest Rate: <%= annualInterestRate %><br />
19    Number of Years: <%= numberOfYears %><br />
20    <b>Monthly Payment:  <%= monthlyPayment %><br />
21    Total Payment: <%= totalPayment %><br /></b>
22    </body>
23  </html>
```

ComputeLoan.html is displayed first to prompt the user to enter the loan amount, annual interest rate, and number of years. Since this file does not contain any JSP elements, it is named with an .html extension as a regular HTML file.

ComputeLoan.jsp is invoked upon clicking the *Compute Loan Payment* button in the HTML form. The JSP page obtains the parameter values using the predefined variable request in lines 7–12 and computes monthly payment and total payment in lines 13–16. The formula for computing monthly payment is given in Listing 2.9, ComputeLoan.java.

What is wrong if the JSP scriptlet <% in line 7 is replaced by the JSP declaration <%!? The predefined variables (e.g., request, response, and out) correspond to local variables defined in the servlet methods doGet and doPost. They must appear in JSP scriptlets, not in JSP declarations.

Tip

ComputeLoan.jsp can also be invoked using the following query string: http://localhost:8084/liangweb/ComputeLoan.jsp?loanAmount=10000&annualInterestRate=6&numberOfYears=15.

Check Point

38.5.1 Describe the predefined variables in JSP.

38.5.2 What is wrong if the JSP scriptlet <% in line 7 in ComputeLoan.jsp (Listing 38.3) is replaced by JSP declaration <%!?

38.5.3 Can you use predefined variables (e.g., request, response, and out) in JSP declarations?

38.6 JSP Directives

Key Point

You can use JSP directives to instruct JSP engine on how to process the JSP code.

A JSP directive is a statement that gives the JSP engine information about the JSP page. For example, if your JSP page uses a Java class from a package other than the java.lang package, you have to use a directive to import this package. The general syntax for a JSP directive is shown below:

```
<%@ directive attribute = "value" %>, or
<%@ directive attribute1 = "value1"
              attribute2 = "value2"
              ...
              attributen = "valuen" %>
```

The possible directives are:

- page lets you provide information for the page, such as importing classes and setting up content type. The page directive can appear anywhere in the JSP file.
- include lets you insert a file into the servlet when the page is translated to a servlet. The include directive must be placed where you want the file to be inserted.
- taglib lets you define custom tags.

The following are useful attributes for the page directive:

- import specifies one or more packages to be imported for this page. For example, the directive <%@ page import="java.util.*, java.text.*" %> imports java.util.* and java.text.*.
- contentType specifies the content type for the resultant JSP page. By default, the content type is text/html for JSP. The default content type for servlets is text/plain.
- session specifies a boolean value to indicate whether the page is part of the session. By default, session is true.
- buffer specifies the output stream buffer size. By default, it is 8KB. For example, the directive <%@ page buffer="10KB" %> specifies that the output buffer size is 10KB. The directive <%@ page buffer="none" %> specifies that a buffer is not used.

- autoFlush specifies a boolean value to indicate whether the output buffer should be automatically flushed when it is full or whether an exception should be raised when the buffer overflows. By default, this attribute is true. In this case, the buffer attribute cannot be none.
- isThreadSafe specifies a boolean value to indicate whether the page can be accessed simultaneously without data corruption. By default, it is true. If it is set to false, the JSP page will be translated to a servlet that implements the SingleThreadModel interface.
- errorPage specifies a JSP page that is processed when an exception occurs in the current page. For example, the directive <%@ page errorPage="HandleError.jsp" %> specifies that HandleError.jsp is processed when an exception occurs.
- isErrorPage specifies a boolean value to indicate whether the page can be used as an error page. By default, this attribute is false.

Listing 38.4 gives an example that shows how to use the page directive to import a class. The example uses the Loan class created in Listing 10.2, Loan.java, to simplify Listing 38.3, ComputeLoan.jsp. You can create an object of the Loan class and use its monthlyPayment() and totalPayment() methods to compute the monthly payment and total payment.

LISTING 38.4 ComputeLoan1.jsp

```
 1  <!-- ComputeLoan1.jsp -->
 2  <html>
 3    <head>
 4      <title>ComputeLoan Using the Loan Class</title>
 5    </head>
 6  <body>
 7    <%@ page import = "chapter38.Loan" %>
 8     <% double loanAmount = Double.parseDouble(
 9          request.getParameter("loanAmount"));
10        double annualInterestRate = Double.parseDouble(
11          request.getParameter("annualInterestRate"));
12        int numberOfYears = Integer.parseInt(
13          request.getParameter("numberOfYears"));
14        Loan loan =
15          new Loan(annualInterestRate, numberOfYears, loanAmount);
16    %>
17    Loan Amount: <%= loanAmount %><br />
18    Annual Interest Rate: <%= annualInterestRate %><br />
19    Number of Years: <%= numberOfYears %><br />
20    <b>Monthly Payment: <%= loan.getMonthlyPayment() %><br />
21    Total Payment: <%= loan.getTotalPayment() %><br /></b>
22    </body>
23  </html>
```

This JSP uses the Loan class. You need to create the class in the liangweb project in package chapter38 as follows:

```
package chapter38;
public class Loan {
  // Same as lines 2–71 in Listing 10.2, Loan.java, so omitted
```

The directive <%@ page import ="chapter38.Loan" %> imports the Loan class in line 7. Line 14 creates an object of Loan for the given loan amount, annual interest rate, and number of years. Lines 20–21 invoke the Loan object's monthlyPayment() and totalPayment() methods to display monthly payment and total payment.

38.6.1 Describe the JSP directives and attributes for the page directive.

38.6.2 If a class does not have a package statement, can you import it?

38.6.3 If you use a custom class from a JSP, where should the class be placed?

38.7 Using JavaBeans in JSP

You can use JavaBeans to create objects for sharing among different JSP pages.

Normally you create an instance of a class in a program and use it in that program. This method is for sharing the class, not the object. JSP allows you to share the object of a class among different pages. To enable an object to be shared, its class must be a JavaBeans component. Recall that this entails the following three features:

- The class is public.
- The class has a public constructor with no arguments.
- The class is serializable. (This requirement is not necessary in JSP.)

To create an instance for a JavaBeans component, use the following syntax:

```
<jsp:useBean id = "objectName" scope = "scopeAttribute"
class = "ClassName" />
```

This syntax is roughly equivalent to

```
<% ClassName objectName = new ClassName() %>
```

except that the scope attribute is missing. The scope attribute specifies the scope of the object, and the object is not recreated if it is already within the scope. Listed below are four possible values for the scope attribute:

- application specifies that the object is bound to the application. The object can be shared by all sessions of the application.
- session specifies that the object is bound to the client's session. Recall that a client's session is automatically created between a Web browser and a Web server. When a client from the same browser accesses two servlets or two JSP pages on the same server, the session is the same.
- page is the default scope, which specifies that the object is bound to the page.
- request specifies that the object is bound to the client's request.

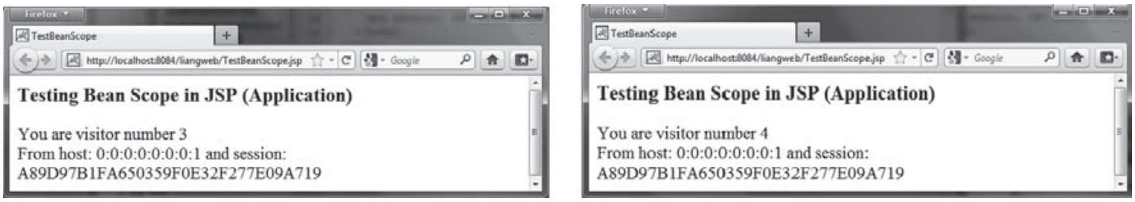
When <jsp:useBean id="objectName" scope="scopeAttribute" class="ClassName" /> is processed, the JSP engine first searches for an object of the class with the same id and scope. If found, the preexisting bean is used; otherwise, a new bean is created.

Here is another syntax for creating a bean:

```
<jsp:useBean id = "objectName" scope = "scopeAttribute"
  class = "ClassName" >
    statements
</jsp:useBean>
```

The statements are executed when the bean is created. If a bean with the same ID and class name already exists in the scope, the statements are not executed.

Listing 38.5 creates a JavaBeans component named Count and uses it to count the number of visits to a JSP page, as shown in Figure 38.7.



Testing Bean Scope in JSP (Application)

You are visitor number 3
From host: 0:0:0:0:0:0:0:1 and session:
A89D97B1FA650359F0E32F277E09A719

Testing Bean Scope in JSP (Application)

You are visitor number 4
From host: 0:0:0:0:0:0:0:1 and session:
A89D97B1FA650359F0E32F277E09A719

FIGURE 38.7 The number of visits to the page is increased when the page is visited.

LISTING 38.5 Count.java

```java
package chapter38;

public class Count {
  private int count = 0;

  /** Return count property */
  public int getCount() {
    return count;
  }

  /** Increase count */
  public void increaseCount() {
   count++;
  }
}
```

The JSP page named TestBeanScope.jsp is created in Listing 38.6.

LISTING 38.6 TestBeanScope.jsp

```
 1  <-- TestBeanScope.jsp -->
 2  <%@ page import = "chapter38.Count" %>
 3  <jsp:useBean id = "count" scope = "application"
 4    class = "chapter38.Count">
 5  </jsp:useBean>
 6  <html>
 7    <head>
 8      <title>TestBeanScope</title>
 9    </head>
10    <body>
11      <h3>Testing Bean Scope in JSP (Application)</h3>
12      <% count.increaseCount(); %>
13      You are visitor number <%= count.getCount() %><br />
14      From host: <%= request.getRemoteHost() %>
15      and session: <%= session.getId() %>
16    </body>
17  </html>
```

The `scope` attribute specifies the scope of the bean. `scope="application"` (line 3) specifies that the bean is alive in the JSP engine and available for all clients to access. The bean can be shared by any client with the directive `<jsp:useBean id="count" scope="application" class="Count">` (lines 3–4). Every client accessing TestBeanScope.jsp causes the count to increase by `1`. The first client causes `count` object to be created, and subsequent access to `TestBeanScope` uses the same object.

If `scope="application"` is changed to `scope="session"`, the scope of the bean is limited to the session from the same browser. The count will increase only if the page is requested from the same browser. If `scope="application"` is changed to `scope="page"`, the scope of the bean is limited to the page, and any other page cannot access this bean. The page will always display count 1. If scope="application" is changed to scope="request", the scope of the bean is limited to the client's request, and any other request on the page will always display count 1.

If the page is destroyed, the count restarts from 0. You can fix the problem by storing the count in a random access file or in a database table. Assume you store the count in the Count table in a database. The Count class can be modified in Listing 38.7.

Listing 38.7 Count.java (Revised Version)

```
package chapter38;

import java.sql.*;

public class Count {
  private int count = 0;
  private Statement statement = null;

  public Count() {
    initializeJdbc();
  }

  /** Return count property */
  public int getCount() {
    try {
      ResultSet rset = statement.executeQuery
        ("select countValue from Count");
      rset.next();
      count = rset.getInt(1);
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }

    return count;
  }

  /** Increase count */
  public void increaseCount() {
    count++;
    try {
      statement.executeUpdate(
        "update Count set countValue = " + count);
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  /** Initialize database connection */
  public void initializeJdbc() {
    try {
      Class.forName("com.mysql.jdbc.Driver");

      // Connect to the sample database
      Connection connection = DriverManager.getConnection
        ("jdbc:mysql://localhost/javabook" , "scott", "tiger");

      statement = connection.createStatement();
    }
```

```
      catch (Exception ex) {
        ex.printStackTrace();
      }
    }
  }
```

38.7.1 You can create an object in a JSP scriptlet. What is the difference between an object created using the `new` operator and a bean created using the `<jsp:useBean ... >` tag?

38.7.2 What is the `scope` attribute for? Describe four scope attributes.

38.7.3 Describe how a `<jsp:useBean ... >` statement is processed by the JSP engine.

38.8 Getting and Setting Properties

JSP provides convenient syntax for getting and setting JavaBeans properties.

By convention, a JavaBeans component provides the get and set methods for reading and modifying its private properties. You can get the property in JSP using the syntax shown below:

```
<jsp:getProperty name = "beanId" property = "sample" />
```

This is roughly equivalent to

```
<%= beanId.getSample() %>
```

You can set the property in JSP using the following syntax:

```
<jsp:setProperty name = "beanId"
  property = "sample" value = "test1" />
```

This is equivalent to

```
<% beanId.setSample("test1"); %>
```

38.9 Associating Properties with Input Parameters

Often properties are associated with input parameters. Suppose you want to get the value of the input parameter named `score` and set it to the JavaBeans property named `score`. You could write the following code:

```
<% double score = Double.parseDouble(
     request.getParameter("score")); %>
<jsp:setProperty name = "beanId"  property = "score"
     value = "<%= score %>" />
```

This is cumbersome. JSP provides a convenient syntax that can be used to simplify it:

```
<jsp:setProperty name = "beanId" property = "score"
   param = "score" />
```

Instead of using the `value` attribute, you use the `param` attribute to name an input parameter. The value of this parameter is set to the property.

Note

Simple type conversion is performed automatically when a bean property is associated with an input parameter. A string input parameter is converted to an appropriate primitive data type or a wrapper class for a primitive type. For example, if the bean property is of the `int` type, the value of the parameter will be converted to the `int` type. If the bean property is of the `Integer` type, the value of the parameter will be converted to the `Integer` type.

Often the bean property and the parameter have the same name. You can use the following convenient statement to associate all the bean properties in `beanId` with the parameters that match the property names:

```
<jsp:setProperty name = "beanId" property = "*" />
```

38.9.1 Example: Computing Loan Payments Using JavaBeans

This example uses JavaBeans to simplify Listing 38.4, ComputeLoan1.jsp, by associating the bean properties with the input parameters. The new ComputeLoan2.jsp is given in Listing 38.8.

LISTING 38.8 ComputeLoan2.jsp

```
1  <!-- ComputeLoan2.jsp -->
2  <html>
3    <head>
4      <title>ComputeLoan Using the Loan Class</title>
5    </head>
6    <body>
7      <%@ page import = "chapter38.Loan" %>
8      <jsp:useBean id = "loan" class = "chapter38.Loan"
9        scope = "page" ></jsp:useBean>
10     <jsp:setProperty name = "loan" property = "*" />
11     Loan Amount: <%= loan.getLoanAmount() %><br />
12     Annual Interest Rate: <%= loan.getAnnualInterestRate() %><br />
13     Number of Years: <%= loan.getNumberOfYears() %><br />
14     <b>Monthly Payment: <%= loan.monthlyPayment() %><br />
15     Total Payment: <%= loan.totalPayment() %><br /></b>
16   </body>
17 </html>
```

Lines 8–9

```
<jsp:useBean id = "loan" class = "chapter38.Loan"
  scope = "page" ></jsp:useBean>
```

create a bean named `loan` for the `Loan` class. Line 10

```
<jsp:setProperty name = "loan" property = "*" />
```

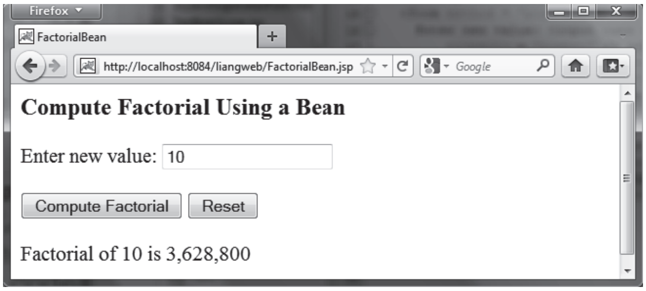
associates the bean properties `loanAmount`, `annualInteresteRate`, and `numberOfYears` with the input parameter values and performs type conversion automatically.

Lines 11–13 use the accessor methods of the loan bean to get the loan amount, annual interest rate, and number of years.

This program acts the same as in Listings 38.3 and 38.4, `ComputeLoan.jsp` and `ComputeLoan1.jsp`, but the coding is much, more simplified.

38.9.2 Example: Computing Factorials Using JavaBeans

This example creates a JavaBeans component named `FactorialBean` and uses it to compute the factorial of an input number in a JSP page named FactorialBean.jsp, as shown in Figure 38.8.



Compute Factorial Using a Bean

Enter new value: 10

Compute Factorial Reset

Factorial of 10 is 3,628,800

FIGURE 38.8 The factorial of an input integer is computed using a method in `FactorialBean`.

Create a JavaBeans component named `FactorialBean.java` (Listing 38.9). Create FactorialBean.jsp (Listing 38.10).

LISTING 38.9 FactorialBean.java

```java
package chapter38;

public class FactorialBean {
  private int number;

  /** Return number property */
  public int getNumber() {
    return number;
  }

  /** Set number property */
  public void setNumber(int newValue) {
    number = newValue;
  }

  /** Obtain factorial */
  public long getFactorial() {
    long factorial = 1;
    for (int i = 1; i <= number; i++)
      factorial *= i;
    return factorial;
  }
}
```

LISTING 38.10 FactorialBean.jsp

```jsp
<!-- FactorialBean.jsp -->
<%@ page import = "chapter38.FactorialBean" %>
<jsp:useBean id = "factorialBeanId"
  class = "chapter38.FactorialBean" scope = "page" >
</jsp:useBean>
<jsp:setProperty name = "factorialBeanId" property = "*" />
<html>
  <head>
    <title>
      FactorialBean
    </title>
  </head>
  <body>
  <h3>Compute Factorial Using a Bean</h3>
```

```
15    <form method = "post">
16      Enter new value: <input name = "number" /><br /><br />
17      <input type = "submit" name = "Submit"
18        value = "Compute Factorial" />
19      <input type = "reset" value = "Reset" /><br /><br />
20      Factorial of
21        <jsp:getProperty name = "factorialBeanId"
22          property = "number" /> is
23        <%@ page import = "java.text.*" %>
24        <% NumberFormat format = NumberFormat.getNumberInstance(); %>
25        <%= format.format(factorialBeanId.getFactorial()) %>
26     </form>
27    </body>
28  </html>
```

The `jsp:useBean` tag (lines 3–4) creates a bean `factorialBeanId` of the `FactorialBean` class. Line 5 `<jsp:setProperty name="factorialBeanId" property="*" />` associates all the bean properties with the input parameters that have the same name. In this case, the bean property `number` is associated with the input parameter `number`. When you click the *Compute Factorial* button, JSP automatically converts the input value for `number` from string into `int` and sets it to `factorialBean` before other statements are executed.

Lines 21–22 `<jsp:getProperty name="factorialBeanId" property="number" />` tag (line 21) is equivalent to `<%= factorialBeanId.getNumber() %>`. The method `factorialBeanId.getFactorial()` (line 25) returns the factorial for the number in `factorialBeanId`.

Design Guide

Mixing a lot of Java code with HTML in a JSP page makes the code difficult to read and to maintain. You should move the Java code to a .java file as much as you can.

Following the preceding design guide, you may improve the preceding example by moving the Java code in lines 23–25 to the `FactorialBean` class. The new `FactorialBean.java` and `FactorialBean.jsp` are given in Listings 38.11 and 38.12.

Listing 38.11 NewFactorialBean.java

```
package chapter38;

import java.text.*;

public class NewFactorialBean {
  private int number;

  /** Return number property */
  public int getNumber() {
    return number;
  }

  /** Set number property */
  public void setNumber(int newValue) {
    number = newValue;
  }

  /** Obtain factorial */
  public long getFactorial() {
    long factorial = 1;
    for (int i = 1; i <= number; i++)
      factorial *= i;
    return factorial;
  }

  /** Format number */
  public static String format(long number) {
    NumberFormat format = NumberFormat.getNumberInstance();
    return format.format(number);
  }
}
```

Listing 38.12 NewFactorialBean.jsp

```
<!-- NewFactorialBean.jsp -->
<%@ page import = "chapter38.NewFactorialBean" %>
<jsp:useBean id = "factorialBeanId"
  class = "chapter38.NewFactorialBean" scope = "page" >
</jsp:useBean>
<jsp:setProperty name = "factorialBeanId" property = "*" />
<html>
  <head>
    <title>
      FactorialBean
    </title>
  </head>
  <body>
  <h3>Compute Factorial Using a Bean</h3>
  <form method = "post">
    Enter new value: <input name = "number" /><br /><br />
    <input type = "submit" name = "Submit"
      value = "Compute Factorial" />
    <input type = "reset" value = "Reset" /><br /><br />
      Factorial of
        <jsp:getProperty name = "factorialBeanId"
          property = "number" /> is
        <%= NewFactorialBean.format(factorialBeanId.getFactorial()) %>
    </form>
  </body>
</html>
```

There is a problem in this page. The program cannot display large factorials. For example, if you entered value 21, the program would display an incorrect factorial. To fix this problem, all you need to do is to revise the NewFactorialBean class using BigInteger to computing factorials (see Exercise 38.18).

38.9.3 Example: Displaying International Time

Listing 37.5, TimeForm.java, gives a Java servlet that uses the doGet method to generate an HTML form for the user to specify a locale and time zone (see Figure 37.18a) and uses the doPost method to display the current time for the specified time zone in the specified locale (see Figure 37.18b). This section rewrites the servlet using JSP. You have to create two JSP pages, one for displaying the form, and the other for displaying the current time.

In the TimeForm.java servlet, arrays allLocale and allTimeZone are the data fields. The doGet and doPost methods both use the arrays. Since the available locales and time zones are used in both pages, it is better to create an object that contains all available locales and time zones. This object can be shared by both pages.

Let us create a JavaBeans component named TimeBean.java (Listing 38.13). This class obtains all the available locales in an array in line 7 and all time zones in an array in line 8. The bean properties `localeIndex` and `timeZoneIndex` (lines 9–10) are defined to refer to an element in the arrays. The `currentTimeString()` method (lines 42–52) returns a string for the current time with the specified locale and time zone.

Listing 38.13 TimeBean.java

```
 1  package chapter38;
 2
 3  import java.util.*;
 4  import java.text.*;
 5
 6  public class TimeBean {
 7    private Locale[] allLocale = Locale.getAvailableLocales();
 8    private String[] allTimeZone = TimeZone.getAvailableIDs();
 9    private int localeIndex;
10    private int timeZoneIndex;
11
12    public TimeBean() {
13      Arrays.sort(allTimeZone);
14    }
15
16    public Locale[] getAllLocale() {
17      return allLocale;
18    }
19
20    public String[] getAllTimeZone() {
21      return allTimeZone;
22    }
23
24    public int getLocaleIndex() {
25      return localeIndex;
26    }
27
28    public int getTimeZoneIndex() {
29      return timeZoneIndex;
30    }
31
32    public void setLocaleIndex(int index) {
33      localeIndex = index;
34    }
35
36    public void setTimeZoneIndex(int index) {
37      timeZoneIndex = index;
38    }
39
40    /** Return a string for the current time
41     * with the specified locale and time zone */
42    public String currentTimeString(
43        int localeIndex, int timeZoneIndex) {
44      Calendar calendar =
45        new GregorianCalendar(allLocale[localeIndex]);
46      TimeZone timeZone =
47        TimeZone.getTimeZone(allTimeZone[timeZoneIndex]);
48      DateFormat dateFormat = DateFormat.getDateTimeInstance(
49        DateFormat.FULL, DateFormat.FULL, allLocale[localeIndex]);
50      dateFormat.setTimeZone(timeZone);
51      return dateFormat.format(calendar.getTime());
52    }
53  }
```

Create DisplayTimeForm.jsp (Listing 38.14). This page displays a form just like the one shown in Figure 37.18a. Line 2 imports the TimeBean class. A bean is created in lines 3–5 and is used in lines 17, 19, 24, and 26 to return all locales and time zones. The scope of the bean is application (line 4), so the bean can be shared by all sessions of the application.

Listing 38.14 DisplayTimeForm.jsp

```
1  <!-- DisplayTimeForm.jsp -->
2  <%@ pageimport = "chapter38.TimeBean" %>
3  <jsp:useBean id = "timeBeanId"
4    class = "chapter38.TimeBean" scope = "application" >
5  </jsp:useBean>
6
7  <html>
8    <head>
9      <title>
10       Display Time Form
11     </title>
12   </head>
13   <body>
14   <h3>Choose locale and time zone</h3>
15   <form method = "post" action = "DisplayTime.jsp">
16     Locale <select size = "1" name = "localeIndex">
17     <% for (int i = 0; i < timeBeanId.getAllLocale().length; i++) {%>
18            <option value = "<%= i %>">
19              <%= timeBeanId.getAllLocale()[i] %>
20            </option>
21     <%}%>
22            </select><br />
23     Time Zone <select size = "1" name = "timeZoneIndex">
24     <% for (int i = 0; i < timeBeanId.getAllTimeZone().length; i++) {%>
25            <option value = "<%= i %>">
26              <%= timeBeanId.getAllTimeZone()[i] %>
27            </option>
28         <%}%>
29         </select><br />
30     <input type = "submit" name = "Submit"
31       value = "Get Time" />
32     <input type = "reset" value = "Reset" />
33     </form>
34   </body>
35 </html>
```

Create DisplayTime.jsp (Listing 38.15). This page is invoked from DisplayTimeForm.jsp to display the time with the specified locale and time zone, just as in Figure 37.18b.

Listing 38.15 DisplayTime.jsp

```
<!-- DisplayTime.jsp -->
<%@page pageEncoding = "GB18030"%>
<%@ page import = "chapter38.TimeBean" %>
<jsp:useBean id = "timeBeanId"
  class = "chapter38.TimeBean" scope = "application" >
</jsp:useBean>
<jsp:setProperty name = "timeBeanId" property = "*" />

<html>
  <head>
```

```
11      <title>
12        Display Time
13      </title>
14    </head>
15    <body>
16    <h3>Choose locale and time zone</h3>
17      Current time is
18        <%= timeBeanId.currentTimeString(timeBeanId.getLocaleIndex(),
19          timeBeanId.getTimeZoneIndex()) %>
20    </body>
21  <html>
```

Line 2 sets the character encoding for the page to GB18030 for displaying international characters. By default, it is UTF-8.

Line 5 imports `chapter38.TimeBean` and creates a bean using the same id as in the preceding page. Since the object is already created in the preceding page, the `timeBeanId` in this page (lines 4–6) and in the preceding page point to the same object.

38.9.4 Example: Registering Students

Listing 37.11, RegistrationWithHttpSession.java, gives a Java servlet that obtains student information from an HTML form (see Figure 37.21) and displays the information for user confirmation (see Figure 37.22). Once the user confirms it, the servlet stores the data into the database. This section rewrites the servlet using JSP. You will create two JSP pages, one named GetRegistrationData.jsp for displaying the data for user confirmation and the other named StoreData.jsp for storing the data into the database.

Since every session needs to connect to the same database, you should declare a class for connecting to the database and for storing a student to the database. This class named `StoreData` is given in Listing 38.16. The `initializeJdbc` method (lines 15–31) connects to the database and creates a prepared statement for storing a record to the Address table. The `storeStudent` method (lines 34–45) executes the prepared statement to store a student address. The `Address` class is created in Listing 37.12.

LISTING 38.16 StoreData.java

```
 1  package chapter38;
 2
 3  import java.sql.*;
 4  import chapter37.Address;
 5
 6  public class StoreData {
 7    // Use a prepared statement to store a student into the database
 8    private PreparedStatement pstmt;
 9
10    public StoreData() {
11      initializeJdbc();
12    }
13
14    /** Initialize database connection */
15    private void initializeJdbc() {
16      try {
17        Class.forName("com.mysql.jdbc.Driver");
18
19        // Connect to the sample database
20        Connection connection = DriverManager.getConnection
21          ("jdbc:mysql://localhost/javabook" , "scott", "tiger");
22
```

```
      // Create a Statement
      pstmt = connection.prepareStatement("insert into Address " +
        "(lastName, firstName, mi, telephone, email, street, city, "
        + "state, zip) values (?, ?, ?, ?, ?, ?, ?, ?, ?)");
    }
    catch (Exception ex) {
      System.out.println(ex);
    }
  }

  /** Store a student record to the database */
  public void storeStudent(Address address) throws SQLException {
    pstmt.setString(1, address.getLastName());
    pstmt.setString(2, address.getFirstName());
    pstmt.setString(3, address.getMi());
    pstmt.setString(4, address.getTelephone());
    pstmt.setString(5, address.getEmail());
    pstmt.setString(6, address.getStreet());
    pstmt.setString(7, address.getCity());
    pstmt.setString(8, address.getState());
    pstmt.setString(9, address.getZip());
    pstmt.executeUpdate();
  }
}
```

The HTML file that displays the form is identical to `Registration.html` in Listing 37.8 except that the action is replaced by HGetRegistrationData.jsp.

`GetRegistrationData.jsp`, which obtains the data from the form, is shown in Listing 38.17. A bean is created in lines 3–4. Line 5 obtains the property values from the form. This is a shorthand notation. Note the parameter names and the property names must be the same to use this notation.

Listing 38.17 GetRegistrationData.jsp

```
 1  <!-- GetRegistrationData.jsp -->
 2  <%@ page import = "chapter37.Address" %>
 3  <jsp:useBean id = "addressId"
 4    class = "chapter37.Address" scope = "session"></jsp:useBean>
 5  <jsp:setProperty name = "addressId" property = "*" />
 6
 7  <html>
 8    <body>
 9      <h1>Registration Using JSP</h1>
10
11      <%
12      if (addressId.getLastName() == null ||
13          addressId.getFirstName() == null) {
14        out.println("Last Name and First Name are required");
15        return; // End the method
16      }
17      %>
18
19      <p>You entered the following data</p>
20      <p>Last name: <%= addressId.getLastName() %></p>
21      <p>First name: <%= addressId.getFirstName() %></p>
22      <p>MI: <%= addressId.getMi() %></p>
23      <p>Telephone: <%= addressId.getTelephone() %></p>
```

```
24      <p>Email: <%= addressId.getEmail() %></p>
25      <p>Address: <%= addressId.getStreet() %></p>
26      <p>City: <%= addressId.getCity() %></p>
27      <p>State: <%= addressId.getState() %></p>
28      <p>Zip:  <%= addressId.getZip() %></p>
29
30      <!-- Set the action for processing the answers -->
31      <form method = "post" action = "StoreStudent.jsp">
32        <input type = "submit" value = "Confirm">
33      </form>
34    </body>
35  </html>
```

GetRegistrationData.jsp invokes **StoreStudent.jsp** (line 31) when the user clicks the *Confirm* button. In Listing 38.18, the same **addressId** is shared with the preceding page within the scope of the same session in lines 3–4. A bean for **StoreData** is created in lines 5–6 with the scope of application.

Listing 38.18 StoreStudent.jsp

```
1   <!-- StoreStudent.jsp -->
2   <%@ page import = "chapter37.Address" %>
3   <jsp:useBean id = "addressId" class = "chapter37.Address"
4     scope = "session"></jsp:useBean>
5   <jsp:useBean id = "storeDataId" class = "chapter38.StoreData"
6     scope = "application"></jsp:useBean>
7
8   <html>
9     <body>
10      <%
11        storeDataId.storeStudent(addressId);
12
13        out.println(addressId.getFirstName() + " " +
14          addressId.getLastName() +
15          " is now registered in the database");
16        out.close(); // Close stream
17      %>
18    </body>
19  </html>
```

Note

The scope for **addressId** is *session*, but the scope for **storeDataId** is *application*. Why? GetRegistrationData.jsp obtains student information, and StoreData.jsp stores the information in the same session. So the *session* scope is appropriate for **addressId**. All the sessions access the same database and use the same prepared statement to store data. With the *application* scope for **storeDataId**, the bean for **StoreData** needs to be created just once.

Note

The **storeStudent** method in line 11 may throw a **java.sql.SQLException**. In JSP, you can omit the try-block for checked exceptions. In case of an exception, JSP displays an error page.

Tip
Using beans is an effective way to develop JSP. You should put Java code into a bean as much as you can. The bean not only simplifies JSP programming but also makes code reusable. The bean can also be used to implement persistent sessions.

38.10 Forwarding Requests from JavaServer Pages

You can use the JSP forward tag to jump to navigate to another HTML page.

Web applications developed using JSP generally consist of many pages linked together. JSP provides a forwarding tag in the following syntax that can be used to forward a page to another page:

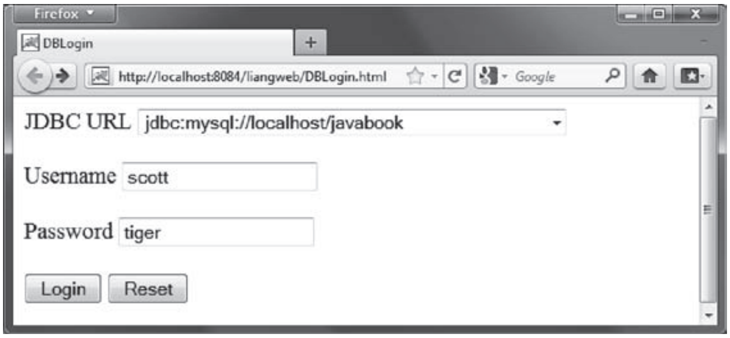
```
<jsp:forward page = "destination" />
```

38.10.1 How do you associate bean properties with input parameters?

38.10.2 How do you write a statement to forward requests to another JSP page?

38.11 Case Study: Browsing Database Tables

This section presents a very useful JSP application for browsing tables. When you start the application, the first page prompts the user to enter the JDBC driver, URL, username, and password for a database, as shown in Figure 38.9. After you log in to the database, you can select a table to browse, as shown in Figure 38.10. Clicking the *Browse Table Content* button displays the table content, as shown in Figure 38.11.



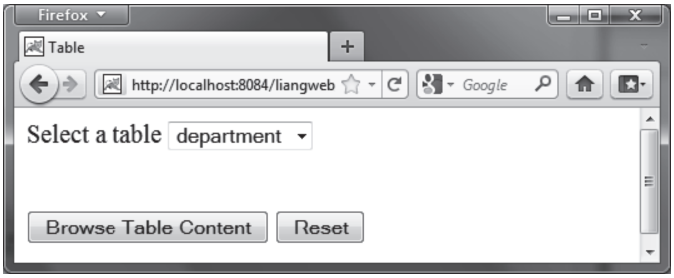
http://localhost:8084/liangweb/DBLogin.html

JDBC URL jdbc:mysql://localhost/javabook

Username scott

Password tiger

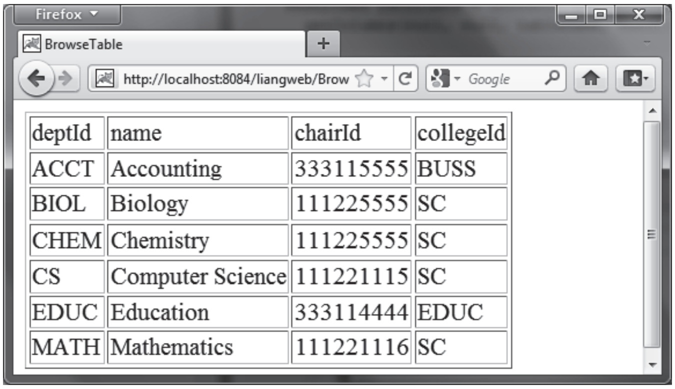
FIGURE 38.9 To access a database, you need to provide the JDBC driver, URL, username, and password.



http://localhost:8084/liangweb

Select a table department

FIGURE 38.10 You can select a table to browse from this page.



deptId	name	chairId	collegeId
ACCT	Accounting	333115555	BUSS
BIOL	Biology	111225555	SC
CHEM	Chemistry	111225555	SC
CS	Computer Science	111221115	SC
EDUC	Education	333114444	EDUC
MATH	Mathematics	111221116	SC

Figure 38.11 The contents of the selected table are displayed.

Create a JavaBeans component named DBBean.java (see Listing 38.19).

Listing 38.19 DBBean.java

```java
package chapter38;

import java.sql.*;

public class DBBean {
  private Connection connection = null;
  private String username;
  private String password;
  private String driver;
  private String url;

  /** Initialize database connection */
  public void initializeJdbc() {
    try {
      System.out.println("Driver is " + driver);
      Class.forName(driver);

      // Connect to the sample database
      connection = DriverManager.getConnection(url, username,
        password);
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  /** Get tables in the database */
  public String[] getTables() {
    String[] tables = null;

    try {
      DatabaseMetaData dbMetaData = connection.getMetaData();
      ResultSet rsTables = dbMetaData.getTables(null, null, null,
        new String[] {"TABLE"});

      int size = 0;
      while (rsTables.next()) size++;

      rsTables = dbMetaData.getTables(null, null, null,
        new String[] {"TABLE"});

      tables = new String[size];
      int i = 0;
      while (rsTables.next())
        tables[i++] = rsTables.getString("TABLE_NAME");
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }

    return tables;
  }

  /** Return connection property */
  public Connection getConnection() {
    return connection;
  }

  public void setUsername(String newUsername) {
    username = newUsername;
  }

  public String getUsername() {
    return username;
  }

  public void setPassword(String newPassword) {
    password = newPassword;
  }

  public String getPassword() {
    return password;
  }

  public void setDriver(String newDriver) {
    driver = newDriver;
  }

  public String getDriver() {
    return driver;
  }

  public void setUrl(String newUrl) {
    url = newUrl;
  }

  public String getUrl() {
    return url;
  }
}
```

Create an HTML file named DBLogin.html (see Listing 38.20) that prompts the user to enter database information and three JSP files named DBLoginInitialization.jsp (see Listing 38.21), Table.jsp (see Listing 38.22), and BrowseTable.jsp (see Listing 38.23) to process and obtain database information.

Listing 38.20 DBLogin.html

```
<!-- DBLogin.html -->
<html>
  <head>
    <title>
      DBLogin
    </title>
  </head>
  <body>
    <form method = "post" action = "DBLoginInitialization.jsp">
    JDBC URL
    <select name = "url" size = "1">
      <option>jdbc:odbc:ExampleMDBDataSource</option>
      <option>jdbc:mysql://localhost/javabook</option>
      <option>jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl</option>
    </select><br /><br />
    Username <input name = "username" /><br /><br />
    Password <input name = "password" /><br /><br />
    <input type = "submit" name = "Submit" value = "Login" />
    <input type = "reset" value = "Reset" />
    </form>
  </body>
</html>
```

Listing 38.21 DBLoginInitialization.jsp

```
<!-- DBLoginInitialization.jsp -->
<%@ page import = "chapter38.DBBean" %>
<jsp:useBean id = "dBBeanId" scope = "session"
  class = "chapter38.DBBean">
</jsp:useBean>
<jsp:setProperty name = "dBBeanId" property = "*" />
<html>
  <head>
    <title>DBLoginInitialization</title>
  </head>
  <body>

  <%-- Connect to the database --%>
  <% dBBeanId.initializeJdbc(); %>

  <% if (dBBeanId.getConnection() == null) { %>
    Error: Login failed. Try again.
  <% }
    else {%>
      <jsp:forward page = "Table.jsp" />
  <% } %>
  </body>
</html>
```

Listing 38.22 Table.jsp

```
<!-- Table.jsp -->
<%@ page import = "chapter38.DBBean" %>
<jsp:useBean id = "dBBeanId" scope = "session"
  class = "chapter38.DBBean">
</jsp:useBean>
<html>
  <head>
```

```
    <title>Table</title>
  </head>
  <body>
  <% String[] tables = dBBeanId.getTables();
    if (tables == null) { %>
      No tables
  <% }
    else { %>
      <form method = "post" action = "BrowseTable.jsp">
        Select a table
         <select name = "tablename" size = "1">
      <% for (int i = 0; i < tables.length; i++) { %>
            <option><%= tables[i] %></option>
      <% }
    } %>
          </select><br /><br /><br />
    <input type = "submit" name = "Submit"
      value = "Browse Table Content">
    <input type = "reset" value = "Reset">
    </form>
  </body>
</html>
```

LISTING 38.23 BrowseTable.jsp

```
<!-- BrowseTable.jsp -->
<%@ page import = "chapter38.DBBean" %>
<jsp:useBean id = "dBBeanId" scope = "session"
  class = "chapter38.DBBean" >
</jsp:useBean>
<%@ page import = "java.sql.*" %>
<html>
  <head>
    <title>BrowseTable</title>
  </head>
  <body>

  <% String tableName = request.getParameter("tablename");

      ResultSet rsColumns = dBBeanId.getConnection().getMetaData().
      getColumns(null, null, tableName, null);
  %>
  <table border = "1">
    <tr>
      <% // Add column names to the table
      while (rsColumns.next()) { %>
        <td><%= rsColumns.getString("COLUMN_NAME") %></td>
    <%}%>
    </tr>

    <% Statement statement =
         dBBeanId.getConnection().createStatement();
       ResultSet rs = statement.executeQuery(
         "select * from " + tableName);

      // Get column count
      int columnCount = rs.getMetaData().getColumnCount();

      // Store rows to rowData
      while (rs.next()) {
```

```
    out.println("<tr>");
    for (int i = 0; i < columnCount; i++) { %>
     <td><%= rs.getObject(i + 1) %></td>
  <%  }
    out.println("</tr>");
   } %>
 </table>
 </body>
</html>
```

You start the application from DBLogin.html. This page prompts the user to enter a JDBC driver, URL, username, and password to log in to a database. A list of accessible drivers and URLs is provided in the selection list. You must make sure that these database drivers are added into the Libraries node in the project.

When you click the *Login* button, DBLoginInitialization.jsp is invoked. When this page is processed for the first time, an instance of `DBBean` named `dBBeanId` is created. The input parameters `driver`, `url`, `username`, and `password` are passed to the bean properties. The `initializeJdbc` method loads the driver and establishes a connection to the database. If login fails, the `connection` property is `null`. In this case, an error message is displayed. If login succeeds, control is forwarded to Table.jsp.

Table.jsp shares `dBBeanId` with DBLoginInitialization.jsp in the same session, so it can access `connection` through `dBBeanId` and obtain tables in the database using the database metadata. The table names are displayed in a selection box in a form. When the user selects a table name and clicks the *Browse Table Content* button, BrowseTable.jsp is processed.

BrowseTable.jsp shares `dBBeanId` with Table.jsp and DBLoginInitialization.jsp in the same session. It retrieves the table contents for the selected table from Table.jsp.

JSP Scripting Constructs Syntax

- `<%= Java expression %>` The expression is evaluated and inserted into the page.
- `<% Java statement %>` Java statements inserted in the `jspService` method.
- `<%! Java declaration %>` Defines data fields and methods.
- `<%-- JSP comment %>` The JSP comments do not appear in the resultant HTML file.
- `<%@ directive attribute="value" %>` The JSP directives give the JSP engine information about the JSP page. For example, `<%@ page import="java.util.*, java.text.*" %>` imports `java.util.*` and `java.text.*`.
- `<jsp:useBean id="objectName" scope="scopeAttribute" class="ClassName" />` Creates a bean if new. If a bean is already created, associates the id with the bean in the same scope.
- `<jsp:useBean id="objectName" scope="scopeAttribute" class="ClassName" > statements </jsp:useBean>` The statements are executed when the bean is created. If a bean with the same id and class name already exists, the statements are not executed.
- `<jsp:getProperty name="beanId" property="sample" />` Gets the property value from the bean, which is the same as `<%= beanId.getSample() %>`.
- `<jsp:setProperty name="beanId" property="sample" value="test1" />` Sets the property value for the bean, which is the same as `<% beanId.setSample("test1"); %>`.
- `<jsp:setProperty name="beanId" property="score" param="score" />` Sets the property with an input parameter.

- `<jsp:setProperty name="beanId" property="*" />` Associates and sets all the bean properties in `beanId` with the input parameters that match the property names.
- `<jsp:forward page="destination" />` Forwards this page to a new page.

JSP Predefined Variables

- `application` represents the `ServletContext` object for storing persistent data for all clients.
- `config` represents the `ServletConfig` object for the page.
- `out` represents the character output stream, which is an instance of `PrintWriter`, obtained from `response.getWriter()`.
- `page` is alternative to `this`.
- `request` represents the client's request, which is an instance of `HttpServletRequest` in the servlet's `service` method.
- `response` represents the client's response, which is an instance of `HttpServletResponse` in the servlet's `service` method.
- `session` represents the `HttpSession` object associated with the request, obtained from `request.getSession()`.

Chapter Summary

1. A JavaServer page is like a regular HTML page with special tags, known as *JSP tags*, which enable the Web server to generate dynamic content. You can create a webpage with static HTML and enclose the code for generating dynamic content in the JSP tags.

2. A JSP page must be stored in a file with a .jsp extension. The Web server translates the .jsp file into a Java servlet, compiles the servlet, and executes it. The result of the execution is sent to the browser for display.

3. A JSP page is translated into a servlet when the page is requested for the first time. It is not retranslated if the page is not modified. To ensure that the first-time real user does not encounter a delay, JSP developers should test the page after it is installed.

4. There are three main types of JSP constructs: scripting constructs, directives, and actions. *Scripting* elements enable you to specify Java code that will become part of the resultant servlet. *Directives* enable you to control the overall structure of the resultant servlet. *Actions* enable you to control the behaviors of the JSP engine.

5. Three types of scripting constructs can be used to insert Java code into the resultant servlet: expressions, scriptlets, and declarations.

6. The scope attribute (application, session, page, and request) specifies the scope of a JavaBeans object. Application specifies that the object be bound to the application. Session specifies that the object be bound to the client’s session. Page is the default scope, which specifies that the object be bound to the page. Request specifies that the object be bound to the client’s request.

7. Web applications developed using JSP generally consist of many pages linked together. JSP provides a forwarding tag in the following syntax that can be used to forward a page to another page: `<jsp:forward page="destination" />`.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

Programming Exercises

MyProgrammingLab™

Note
Solutions to even-numbered exercises in this chapter are in `exercise\jspexercise` from evennumberedexercise.zip, which can be downloaded from the Companion Website.

Section 38.4

38.1 (*Factorial table in JSP*) Rewrite Exercise 37.1 using JSP.

38.2 (*Muliplication table in JSP*) Rewrite Exercise 37.2 using JSP.

Section 38.5

***38.3** (*Obtain parameters in JSP*) Rewrite the servlet in Listing 37.4, GetParameters.java, using JSP. Create an HTML form that is identical to Student_Registration_Form.html in Listing 37.3 except that the action is replaced by `Exercise40_3.jsp` for obtaining parameter values.

Section 38.6

38.4 (*Calculate tax in JSP*) Rewrite Exercise 37.4 using JSP. You need to import ComputeTax in the JSP.

***38.5** (*Find scores from text files*) Rewrite Exercise 37.6 using servlets.

****38.6** (*Find scores from database tables*) Rewrite Exercise 37.7 using servlets.

Section 38.7

****38.7** (*Change the password*) Rewrite Exercise 37.8 using servlets.

Comprehensive

***38.8** (*Store cookies in JSP*) Rewrite Exercise 37.10 using JSP. Use `response.addCookie(Cookie)` to add a cookie.

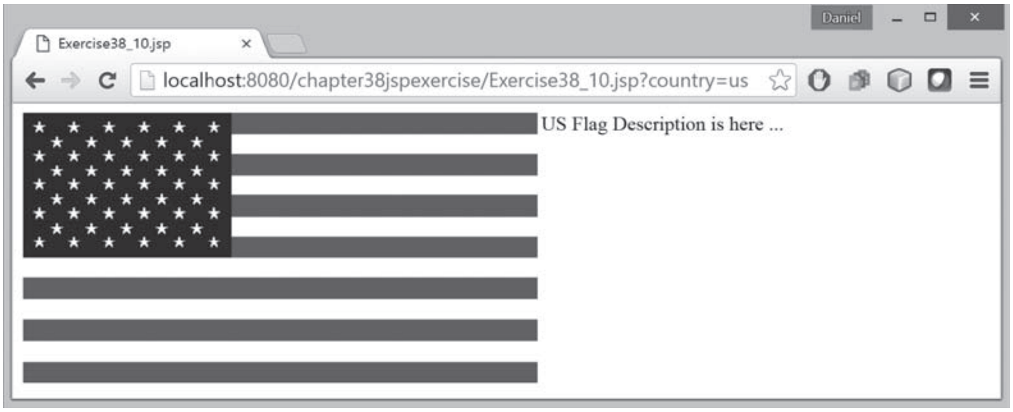
***38.9** (*Retrieve cookies in JSP*) Rewrite Exercise 37.11 using JSP. Use `Cookie[] cookies = request.getCookies()` to get all cookies.

38.10 (*Draw images*) Write a JSP program that displays a country's flag and description as shown in Figure 38.12. The country code such as us is passed as a parameter in the URL. The country's flag file is named as CountryCode.gif and the description is stored in a text file named CountryCode.txt on the server. So, for the country code us, the flag file us.gif and the text file is us.txt.

*****38.11** (*Syntax highlighting*) Rewrite Exercise 37.12 using JSP.

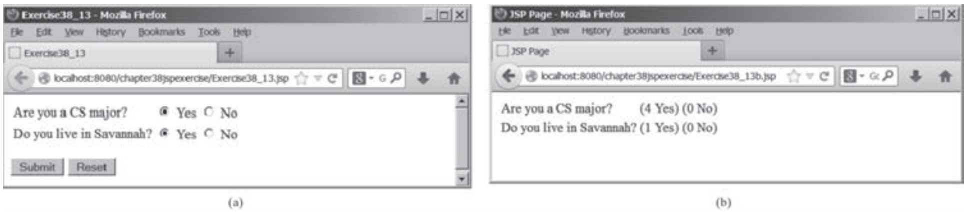
****38.12** (*Opinion poll*) Rewrite Exercise 37.13 using JSP.

*****38.13** (*Multiple-question opinion poll*) The `Poll` table in Exercise 37.13 contains only one question. Suppose you have a `Poll` table that contains multiple questions. Write a JSP that reads all the questions from the table and display them in a form, as shown in Figure 38.13a. When the user clicks the *Submit* button, another JSP page is invoked. This page updates the Yes or No counts for each question and displays the current Yes and No counts for each question in the `Poll` table, as shown in Figure 38.13b. Note that the table may contain many questions. The questions in the figure are just examples. Sort the questions in alphabetical order.


Exercise38_10.jsp
Daniel
localhost:8080/chapter38jspexercise/Exercise38_10.jsp?country=us

US Flag Description is here ...

FIGURE 38.12 The program displays an image and the description of the image.


Exercise38_13 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Exercise38_13
localhost:8080/chapter38jspexercise/Exercise38_13.jsp
Submit
Reset
JSP Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
JSP Page
localhost:8080/chapter38jspexercise/Exercise38_13b.jsp

Are you a CS major? ● Yes ○ No
Do you live in Savannah? ● Yes ○ No

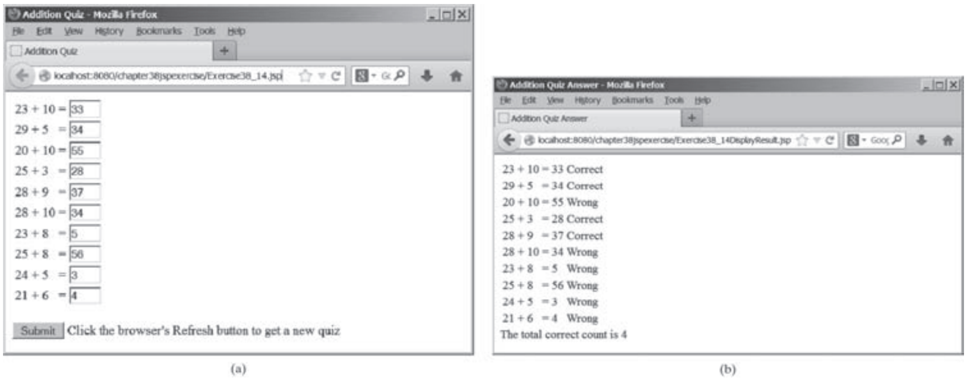
(a)

Are you a CS major? (4 Yes) (0 No)
Do you live in Savannah? (1 Yes) (0 No)

(b)

FIGURE 38.13 The form prompts the user to enter Yes or No for each question in (a), and the updated Yes or No counts are displayed in (b).

38.14 (*Addition quiz*) Write a JSP program that generates addition quizzes randomly, as shown in Figure 38.14a. After the user answers all questions, the JSP displays the result, as shown in Figure 38.14b.


Addition Quiz - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Addition Quiz
localhost:8080/chapter38jspexercise/Exercise38_14.jsp
Submit
Addition Quiz Answer - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Addition Quiz Answer
localhost:8080/chapter38jspexercise/Exercise38_14DisplayResult.jsp

```
23 + 10 = 33
29 + 5  = 34
20 + 10 = 55
25 + 3  = 28
28 + 9  = 37
28 + 10 = 34
23 + 8  = 5
25 + 8  = 56
24 + 5  = 3
21 + 6  = 4

Submit Click the browser's Refresh button to get a new quiz
```

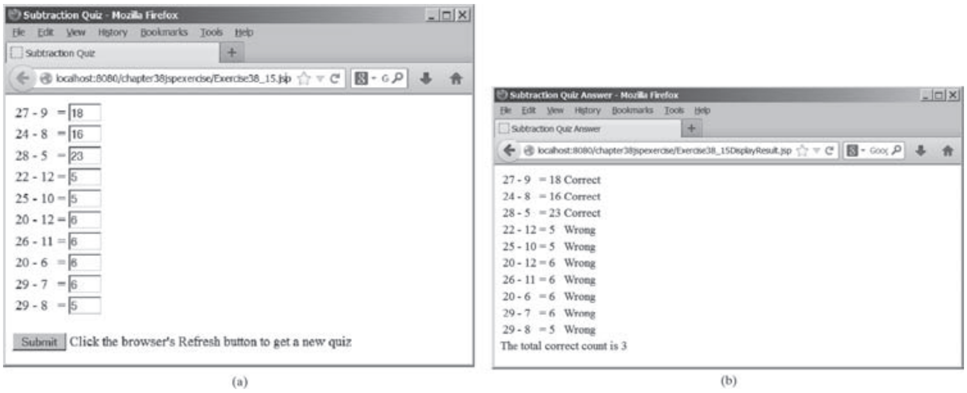
(a)

```
23 + 10 = 33 Correct
29 + 5  = 34 Correct
20 + 10 = 55 Wrong
25 + 3  = 28 Correct
28 + 9  = 37 Correct
28 + 10 = 34 Wrong
23 + 8  = 5  Wrong
25 + 8  = 56 Wrong
24 + 5  = 3  Wrong
21 + 6  = 4  Wrong
The total correct count is 4
```

(b)

FIGURE 38.14 The program displays addition questions in (a) and answers in (b).

38.15 (*Subtraction quiz*) Write a JSP program that generates subtraction quizzes randomly, as shown in Figure 38.14a. The first number must always be greater than or equal to the second number. After the user answers all questions, the JSP displays the result, as shown in Figure 38.14b.


Subtraction Quiz - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Subtraction Quiz
localhost:8080/chapter38jspexercise/Exercise38_15.jsp
Submit
Subtraction Quiz Answer - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Subtraction Quiz Answer
localhost:8080/chapter38jspexercise/Exercise38_15DisplayResult.jsp

```
27 - 9  = 18
24 - 8  = 16
28 - 5  = 23
22 - 12 = 5
25 - 10 = 5
20 - 12 = 6
26 - 11 = 6
20 - 6  = 6
29 - 7  = 6
29 - 8  = 5

Submit  Click the browser's Refresh button to get a new quiz
```

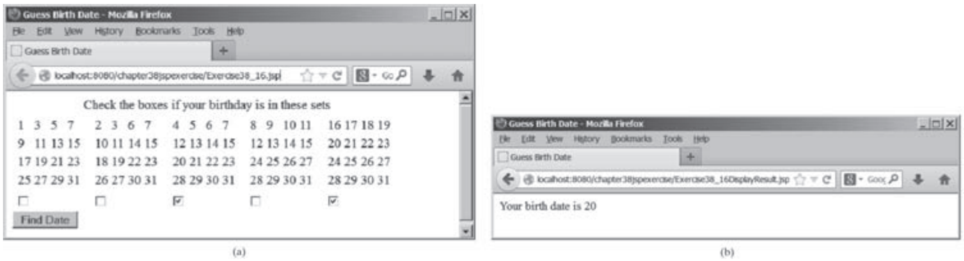
(a)

```
27 - 9  = 18 Correct
24 - 8  = 16 Correct
28 - 5  = 23 Correct
22 - 12 = 5  Wrong
25 - 10 = 5  Wrong
20 - 12 = 6  Wrong
26 - 11 = 6  Wrong
20 - 6  = 6  Wrong
29 - 7  = 6  Wrong
29 - 8  = 5  Wrong
The total correct count is 3
```

(b)

FIGURE 38.14 The program displays subtraction questions in (a) and answers in (b).

****38.16** (*Guess birthday*) Listing 3.3, GuessBirthDay.java, gives a program for guessing a birthday. Write a JSP program that displays five sets of numbers, as shown in Figure 38.15a. After the user checks the appropriate boxes and clicks the *Find Date* button, the program displays the date, as shown in Figure 38.15b.


Guess Birth Date - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Guess Birth Date
localhost:8080/chapter38jspexercise/Exercise38_16.jsp
Find Date
Guess Birth Date - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Guess Birth Date
localhost:8080/chapter38jspexercise/Exercise38_16DisplayResult.jsp

```
Check the boxes if your birthday is in these sets
1  3  5  7    2  3  6  7    4  5  6  7    8  9  10 11   16 17 18 19
9  11 13 15   10 11 14 15   12 13 14 15   12 13 14 15   20 21 22 23
17 19 21 23   18 19 22 23   20 21 22 23   24 25 26 27   24 25 26 27
25 27 29 31   26 27 30 31   28 29 30 31   28 29 30 31   28 29 30 31
□             □             ☑             □             ☑
Find Date
```

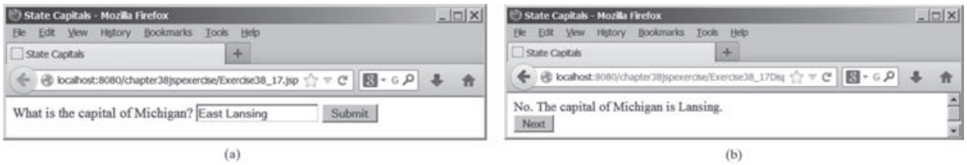
(a)

```
Your birth date is 20
```

(b)

FIGURE 38.15 (a) The program displays five sets of numbers for the user to check the boxes. (b) The program displays the date.

****38.17** (*Guess capitals*) Write a JSP that prompts the user to enter a capital for a state, as shown in Figure 38.16a. Upon receiving the user input, the program reports whether the answer is correct, as shown in Figure 38.16b. You can click the *Next* button to display another question. You can use a two-dimensional array to store the states and capitals, as proposed in Exercise 9.22. Create a list from the array and apply the `shuffle` method to reorder the list so the questions will appear in random order.


State Capitals - Mozilla Firefox
File Edit View History Bookmarks Tools Help
State Capitals
localhost:8080/chapter38jspexercise/Exercise38_17.jsp
Submit
State Capitals - Mozilla Firefox
File Edit View History Bookmarks Tools Help
State Capitals
localhost:8080/chapter38jspexercise/Exercise38_17Dis
Next

```
What is the capital of Michigan? East Lansing  Submit
```

(a)

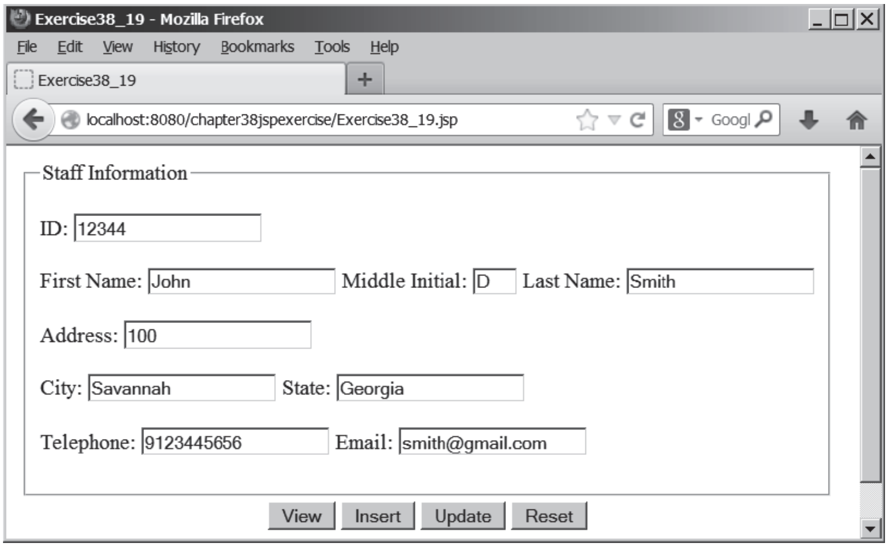
```
No. The capital of Michigan is Lansing.
Next
```

(b)

FIGURE 38.16 (a) The program displays a question. (b) The program displays the answer to the question.

***38.18** (*Large factorial*) Rewrite Listing 38.11 to handle a large factorial. Use the `BigInteger` class introduced in §14.12.

**38.19 (*Access and update a* `Staff` *table*) Write a JSP for Exercise 33.1, as shown in Figure 38.17.


Exercise38_19 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Exercise38_19
localhost:8080/chapter38jspexercise/Exercise38_19.jsp
Googl
View
Insert
Update
Reset

Staff Information

ID: 12344

First Name: John Middle Initial: D Last Name: Smith

Address: 100

City: Savannah State: Georgia

Telephone: 9123445656 Email: smith@gmail.com

Figure 38.17 The JSP page lets you view, insert, and update staff information.

*38.20 (*Guess number*) Write a JSP page that generates a random number between `1` and `1000` and let the user enter a guess. When the user enters a guess, the program should tell the user whether the guess is correct, too high, or too low.

CHAPTER

39

JavaServer Faces

Objectives

- To explain what JSF is (§39.1).
- To create a JSF project in NetBeans (§39.2.1).
- To create a JSF page (§39.2.2).
- To create a JSF managed bean (§39.2.3).
- To use JSF expressions in a facelet (§39.2.4).
- To use JSF GUI components (§39.3).
- To obtain and process input from a form (§39.4).
- To develop a calculator using JSF (§39.5).
- To track sessions in application, session, view, and request scopes (§39.6).
- To validate input using the JSF validators (§39.7).
- To bind database with facelets (§39.8).
- To open a new JSF page from the current page (§39.9).
- To program using contexts and dependency injection (§39.10).

39.1 Introduction

Key Point

JavaServer Faces (JSF) is a new technology for developing server-side Web applications using Java.

JSF

JSF enables you to completely separate Java code from HTML. You can quickly build Web applications by assembling reusable UI components in a page, connecting these components to Java programs and wiring client-generated events to server-side event handlers. The application developed using JSF is easy to debug and maintain.

JSF 2
XHTML
CSS

Note

This chapter introduces JSF 2, the latest standard for JavaServer Faces. You need to know XHTML (eXtensible HyperText Markup Language) and CSS (Cascading Style Sheet) to start this chapter. For information on XHTML and CSS, see Supplements V.A and V.B.

NetBeans 7.3.1
GlassFish 4
Java EE 7

Caution

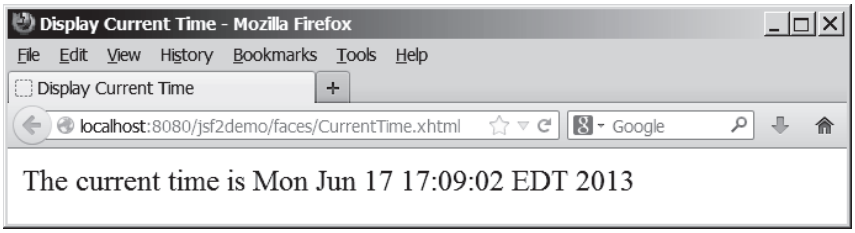
The examples and exercises in this chapter were tested using NetBeans 7.3.1, GlassFish 4, and Java EE 7. You need to use NetBeans 7.3.1 or a higher version with GlassFish 4 and Java EE 7 to develop your JSF projects.

39.2 Getting Started with JSF

Key Point

NetBeans is an effective tool for developing JSF applications.

We begin with a simple example that illustrates the basics of developing JSF projects using NetBeans. The example is to display the date and time on the server, as shown in Figure 39.1.


Display Current Time - Mozilla Firefox
File
Edit
View
History
Bookmarks
Tools
Help
Display Current Time
localhost:8080/jsf2demo/faces/CurrentTime.xhtml
Google

The current time is Mon Jun 17 17:09:02 EDT 2013

FIGURE 39.1 The application displays the date and time on the server.

39.2.1 Creating a JSF Project

Here are the steps to create the application.

create a project

Step 1: Choose *File*, *New Project* to display the New Project dialog box. In this box, choose *Java Web* in the Categories pane and *Web Application* in the Projects pane. Click *Next* to display the New Web Application dialog box.

In the New Web Application dialog box, enter and select the following fields, as shown in Figure 39.2a:

Project Name: `jsf2demo`
Project Location: `c:\book`

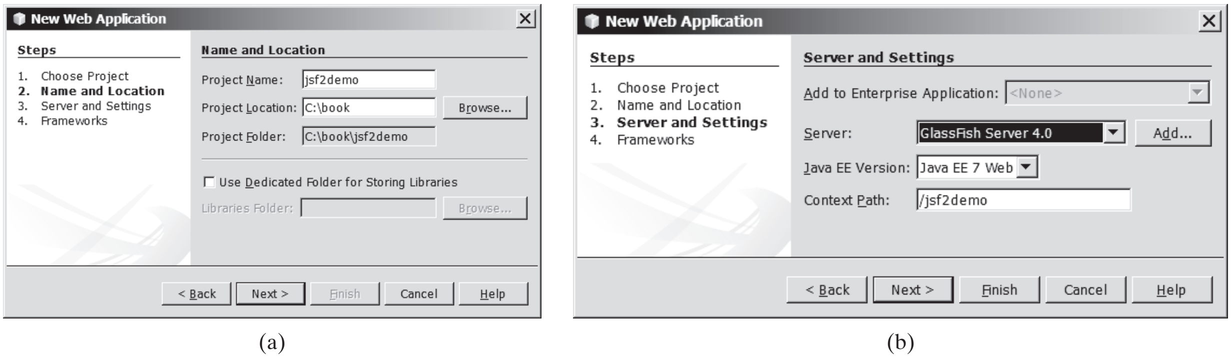
choose server and Java EE 7

Step 2: Click *Next* to display the dialog box for choosing servers and settings. Select the following fields as shown in Figure 39.2b. (Note: You can use any server such as GlassFish 4.x that supports Java EE 6.)

Server: `GlassFish 4`
Java EE Version: `Java EE 7 Web`

Step 3: Click *Next* to display the dialog box for choosing frameworks, as shown in Figure 39.3. Check *JavaServer Faces* and JSF 2.0 as Server Library. Click *Finish* to create the project, as shown in Figure 39.4.

choose JavaServer Faces and JSF2.2



(a)

Steps
1. Choose Project
2. **Name and Location**
3. Server and Settings
4. Frameworks

Name and Location

Project Name: jsf2demo
Project Location: C:\book
Project Folder: C:\book\jsf2demo
□ Use Dedicated Folder for Storing Libraries
Libraries Folder:

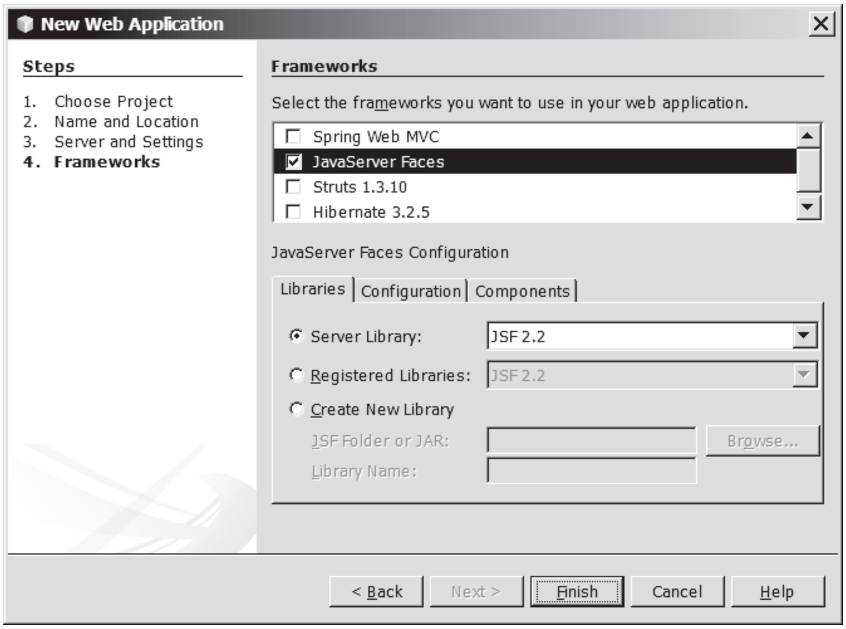
(b)

Steps
1. Choose Project
2. Name and Location
3. **Server and Settings**
4. Frameworks

Server and Settings

Add to Enterprise Application: <None>
Server: GlassFish Server 4.0
Java EE Version: Java EE 7 Web
Context Path: /jsf2demo

FIGURE 39.2 The New Web Application dialog box enables you to create a new Web project.



Steps
1. Choose Project
2. Name and Location
3. Server and Settings
4. **Frameworks**

Frameworks

Select the frameworks you want to use in your web application.

□ Spring Web MVC
☑ JavaServer Faces
□ Struts 1.3.10
□ Hibernate 3.2.5

JavaServer Faces Configuration

● Server Library: JSF 2.2
○ Registered Libraries: JSF 2.2
○ Create New Library
JSF Folder or JAR:
Library Name:

FIGURE 39.3 Check JavaServer Faces and JSF 2.2 to create a Web project.

39.2.2 A Basic JSF Page

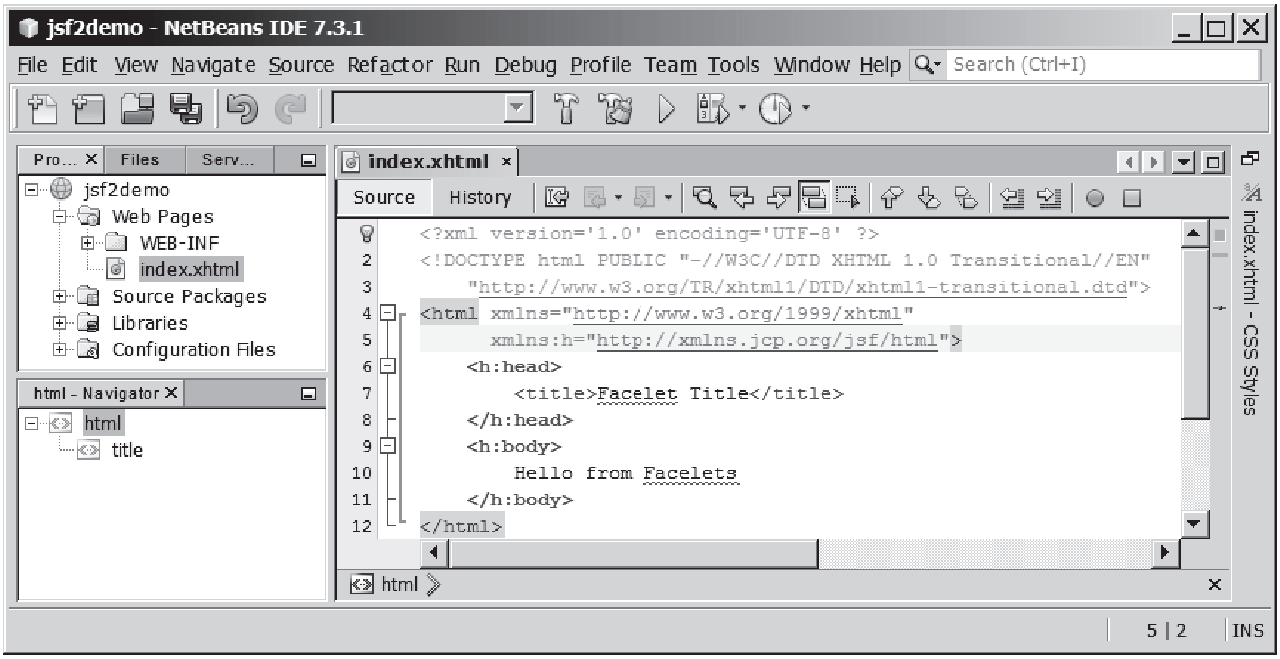
A new project was just created with a default page named index.xhtml, as shown in Figure 39.4. This page is known as a *facelet*, which mixes JSF tags with XHTML tags. Listing 39.1 lists the contents of index.xhtml.

facelet

LISTING 39.1 index.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!-- index.xhtml -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
```

xml version
comment
DOCTYPE
default namespace
JSF namespace



```
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5        xmlns:h="http://xmlns.jcp.org/jsf/html">
6      <h:head>
7          <title>Facelet Title</title>
8      </h:head>
9      <h:body>
10         Hello from Facelets
11     </h:body>
12 </html>
```

FIGURE 39.4 A default JSF page is created in a new Web project.

h:head

h:body

```
 7    <h:head>
 8      <title>Facelet Title</title>
 9    </h:head>
10    <h:body>
11      Hello from Facelets
12    </h:body>
13  </html>
```

XML declaration

Line 1 is an XML declaration to state that the document conforms to the XML version 1.0 and uses the UTF-8 encoding. The declaration is optional, but it is a good practice to use it. A document without the declaration may be assumed of a different version, which may lead to errors. If an XML declaration is present, it must be the first item to appear in the document. This is because an XML processor looks for the first line to obtain information about the document so that it can be processed correctly.

XML comment

Line 2 is a comment for documenting the contents in the file. XML comment always begins with `<!--` and ends with `-->`.

DOCTYPE

Lines 3 and 4 specify the version of XHTML used in the document. This can be used by the Web browser to validate the syntax of the document.

element

tag

An XML document consists of elements described by tags. An element is enclosed between a start tag and an end tag. XML elements are organized in a tree-like hierarchy. Elements may contain subelements, but there is only one root element in an XML document. All the elements must be enclosed inside the root tag. The root element in XHTML is defined using the `html` tag (line 5).

html tag

Each tag in XML must be used in a pair of the start tag and the end tag. A start tag begins with `<` followed by the tag name and ends with `>`. An end tag is the same as its start tag except that it begins with `</`. The start tag and end tag for `html` are `<html>` and `</html>`.

The `html` element is the root element that contains all other elements in an XHTML page. The starting `<html>` tag (lines 5 and 6) may contain one or more `xmlns` (XML namespace) attributes to specify the namespace for the elements used in the document. Namespaces are like Java packages. Java packages are used to organize classes and to avoid naming conflict. XHTML namespaces are used to organize tags and resolve naming conflict. If an element with the same name is defined in two namespaces, the fully qualified tag names can be used to differentiate them.

xmlns

Each xmlns attribute has a name and a value separated by an equal sign (=). The following declaration (line 5)

```
xmlns="http://www.w3.org/1999/xhtml"
```

specifies that any unqualified tag names are defined in the default standard XHTML namespace.

The following declaration (line 6)

```
xmlns:h="http://xmlns.jcp.org/jsf/html"
```

allows the tags defined in the JSF tag library to be used in the document. These tags must have a prefix h.

h:head

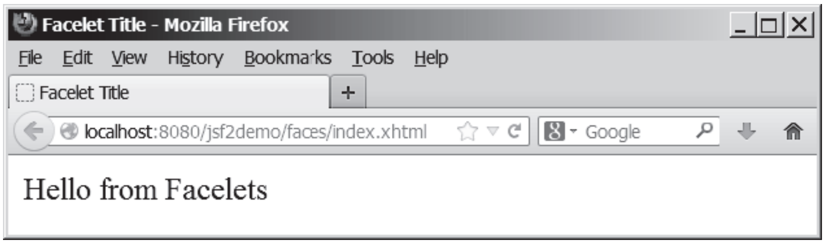
An html element contains a head and a body. The h:head element (lines 7–9) defines an HTML title element. The title is usually displayed in the browser window's title bar.

h:body

An h:body element defines the page's content. In this simple example, it contains a string to be displayed in the Web browser.

Note
The XML tag names are case sensitive, whereas HTML tags are not. So, <html> is different from <HTML> in XML. Every start tag in XML must have a matching end tag, whereas some tags in HTML do not need end tags.

You can now display the page in index.xhtml by right-clicking on index.xhtml in the projects pane and choose *Run File*. The page is displayed in a browser, as shown in Figure 39.5.


Facelet Title - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Facelet Title
+
Google

localhost:8080/jsf2demo/faces/index.xhtml

Hello from Facelets

FIGURE 39.5 The index.xhtml is displayed in the browser.

Note
The JSF page is processed and converted into a regular HTML page for displaying by a browser. The Java software that runs on the server side for producing the HTML page is known as *Java server container* or simply *container*. The container is responsible for handling all server-side tasks for Java EE. GlassFish is a Java server container.

container

39.2.3 Managed JavaBeans for JSF

JSF applications are developed using the Model-View-Controller (MVC) architecture, which separates the application's data (contained in the model) from the graphical presentation (the view). The controller is the JSF framework that is responsible for coordinating interactions between view and the model.

In JSF, the facelets are the view for presenting data. Data are obtained from Java objects. Objects are defined using Java classes. In JSF, the objects that are accessed from a facelet are JavaBeans objects. A *JavaBean* class is simply a public Java class with a no-arg constructor. JavaBeans may contain properties. By convention, a property is defined with a getter and a setter method. If a property only has a getter method, the property is called a read-only property. If a property only has a setter method, the property is called a write-only property. A property does not need to be defined as a data field in the class.

JavaBean

Our example in this section is to develop a JSF facelet to display current time. We will create a JavaBean with a `getTime()` method that returns the current time as a string. The facelet will invoke this method to obtain current time.

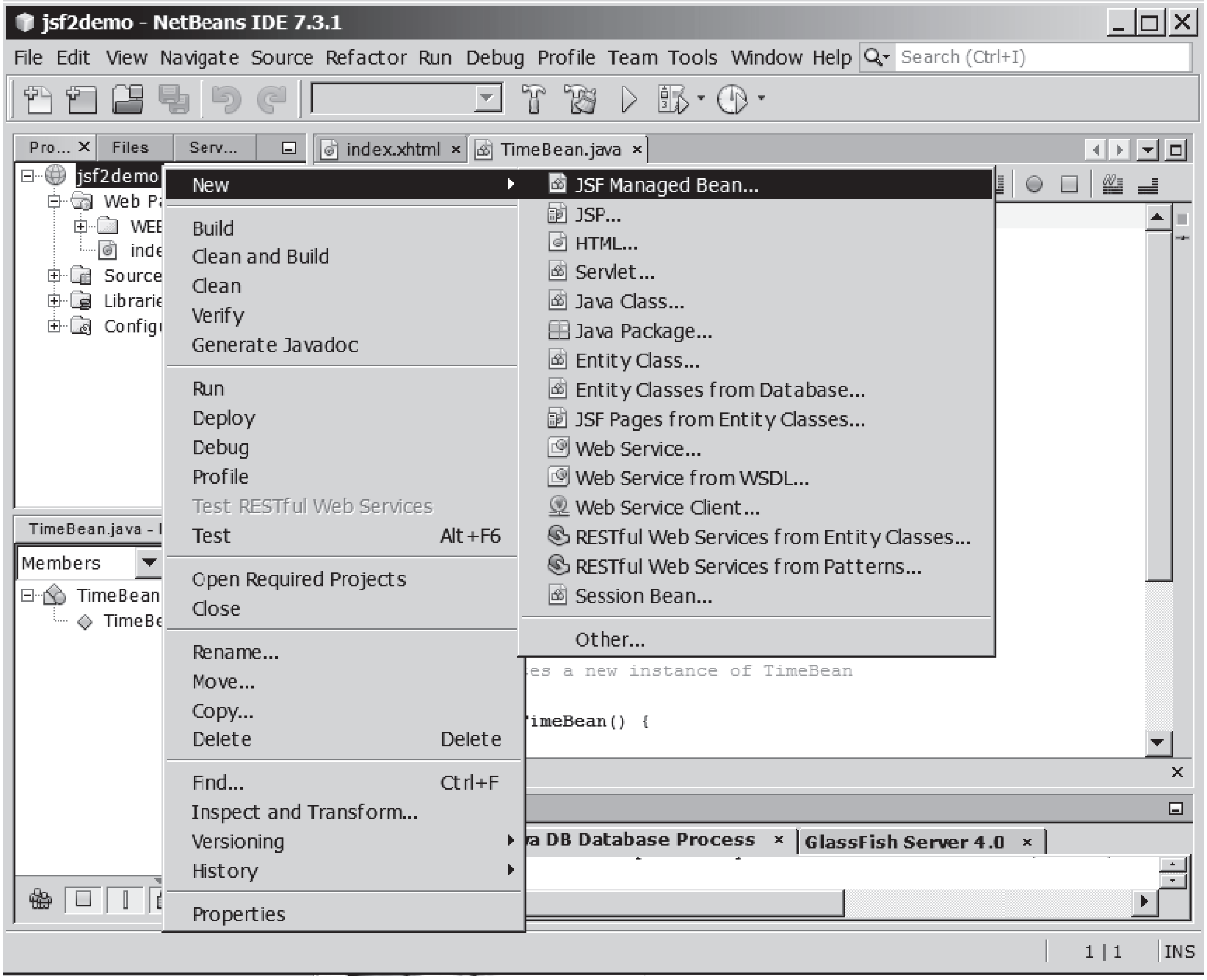
Here are the steps to create a JavaBean named `TimeBean`.

Step 1. Right-click the project node `jsf2demo` to display a context menu as shown in Figure 39.6. Choose *New*, *JSF Managed Bean* to display the New JSF Managed Bean dialog box, as shown in Figure 39.7. (Note: if you don't see JSF Managed Bean in the menu, choose *Other* to locate it in the JavaServer Faces category.)

Step 2. Enter and select the following fields, as shown in Figure 39.7:

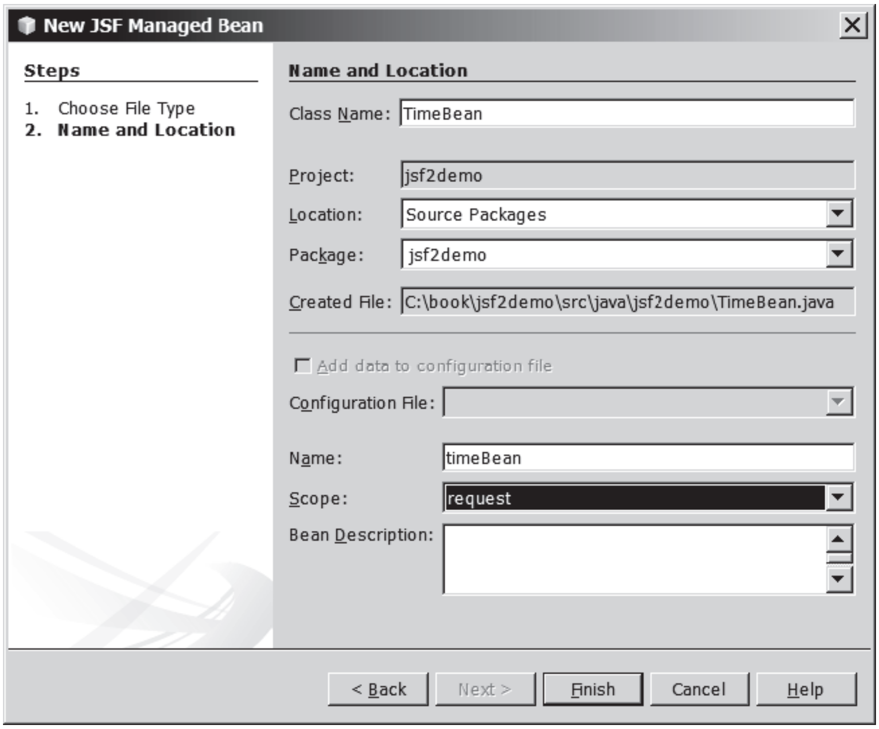
Class Name: `TimeBean`
Package: `jsf2demo`
Name: `timeBean`
Scope: `request`
Click *Finish* to create TimeBean.java, as shown in Figure 39.8.

Step 3. Add the `getTime()` method to return the current time, as shown in Listing 39.2.



```
es a new instance of TimeBean

imeBean() {
```

FIGURE 39.6 Choose JSF Managed Bean to create a JavaBean for JSF.



Class Name: TimeBean

Project: jsf2demo

Location: Source Packages

Package: jsf2demo

Created File: C:\book\jsf2demo\src\java\jsf2demo\TimeBean.java

□ Add data to configuration file

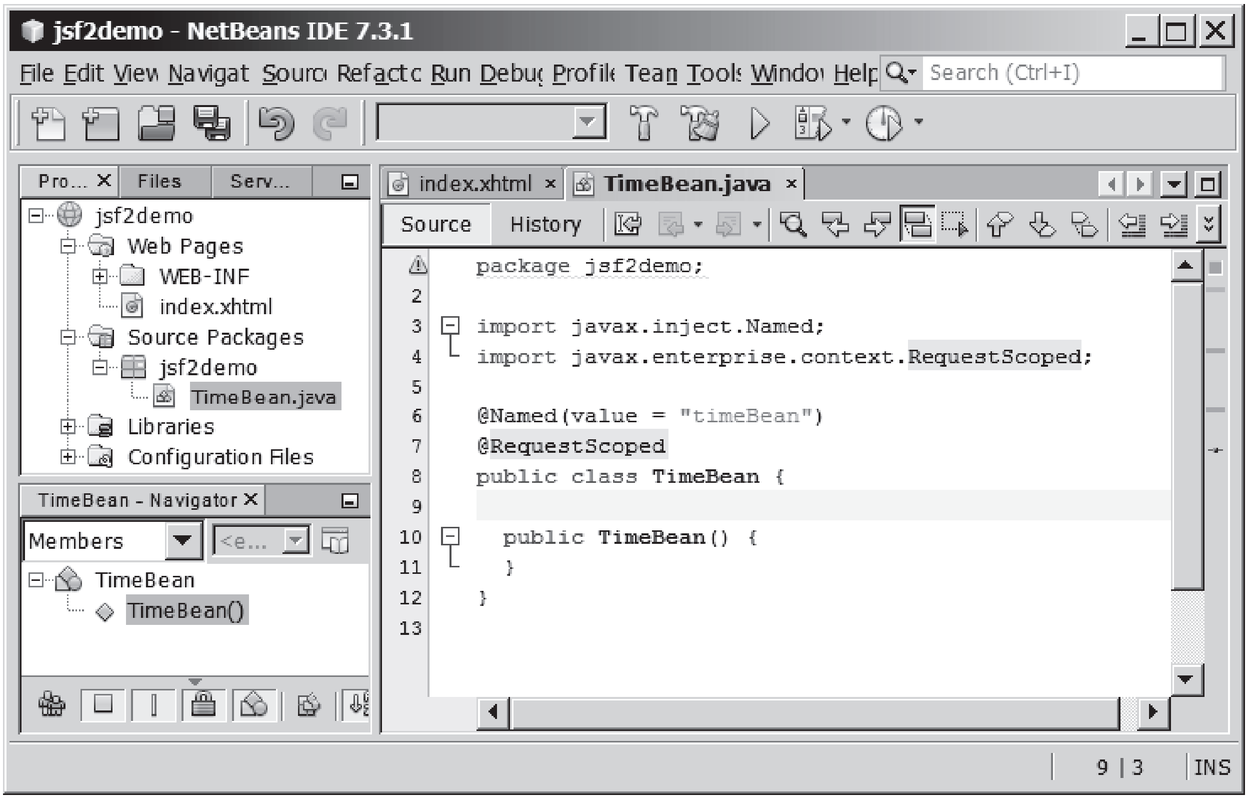
Configuration File:

Name: timeBean

Scope: request

Bean Description:

FIGURE 39.7 Specify the name, location, and scope for the bean.



```
package jsf2demo;

import javax.inject.Named;
import javax.enterprise.context.RequestScoped;

@Named(value = "timeBean")
@RequestScoped
public class TimeBean {

  public TimeBean() {
  }
}
```

FIGURE 39.8 A JavaBean for JSF was created.

Listing 39.2 TimeBean.java

@Named
@RequestScoped
time property

```
package jsf2demo;

import javax.inject.Named;
import javax.enterprise.context.RequestScoped;

@Named (value = "timeBean")
@RequestScoped
public class TimeBean {
  public TimeBean() {
  }

  public String getTime() {
    return new java.util.Date().toString();
  }
}
```

TimeBean is a JavaBeans with the @Named annotation, which indicates that the JSF framework will create and manage the TimeBean objects used in the application. You have learned to use the @Override annotation in Chapter 11. The @Override annotation tells the compiler that the annotated method is required to override a method in a superclass. The @Named annotation tells the compiler to generate the code to enable the bean to be used by JSF facelets.

@RequestScope

The @RequestScope annotation specifies that the scope of the JavaBeans object is within a request. You can also use @ViewScope, @SessionScope or @ApplicationScope to specify the scope for a session or for the entire application.

39.2.4 JSF Expressions

We demonstrate JSF expressions by writing a simple application that displays the current time. You can display current time by invoking the getTime() method in a TimeBean object using a JSF expression.

To keep index.xhtml intact, we create a new JSF page named CurrentTime.xhtml as follows:

Step 1. Right-click the jsf2demo node in the project pane to display a context menu and choose *New*, *JSF Page* to display the New JSF File dialog box, as shown in Figure 39.9.

Step 2. Enter CurrentTime in the File Name field, choose Facelets and click *Finish* to generate CurrentTime.xhtml, as shown in Figure 39.10.

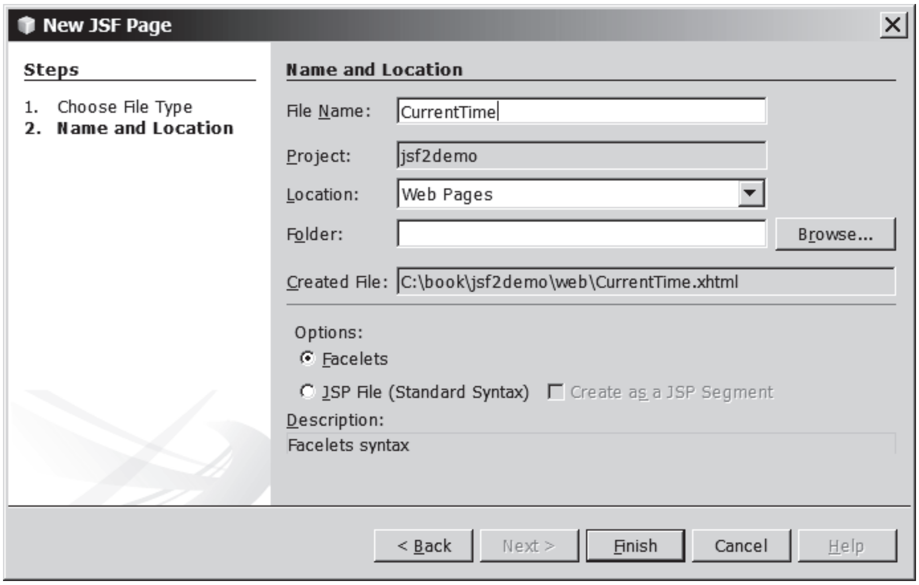
Step 3. Add a JSF expression to obtain the current time, as shown in Listing 39.3.

Step 4. Right-click on CurrentTime.xhtml in the project to display a context menu and choose *Run File* to display the page in a browser as shown in Figure 39.1.

Listing 39.3 CurrentTime.xhtml

refresh page
JSF expression

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
  <h:head>
     <title>Display Current Time</title>
     <meta http-equiv="refresh" content ="60" />
  </h:head>
  <h:body>
    The current time is #{timeBean.time}
  </h:body>
</html>
```



Name and Location

File Name: CurrentTime

Project: jsf2demo

Location: Web Pages

Folder:

Created File: C:\book\jsf2demo\web\CurrentTime.xhtml

Options:

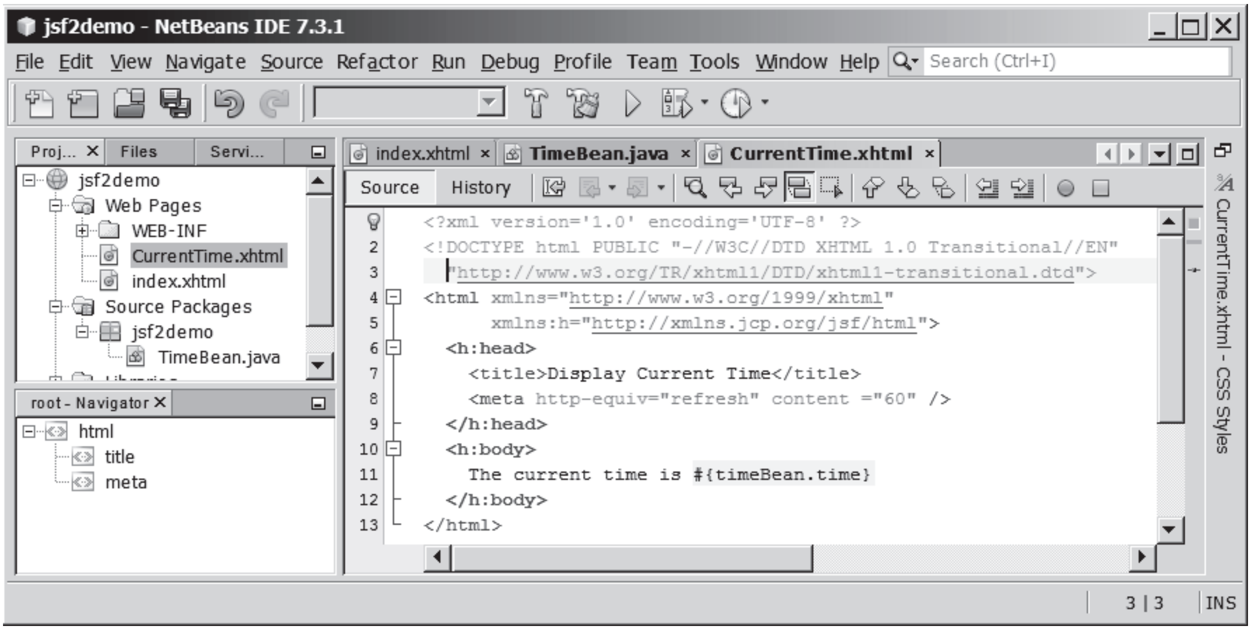
● Facelets

○ JSP File (Standard Syntax) □ Create as a JSP Segment

Description:

Facelets syntax

FIGURE 39.9 The New JSF Page dialog is used to create a JSF page.



```
1   <?xml version='1.0' encoding='UTF-8' ?>
2   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4   <html xmlns="http://www.w3.org/1999/xhtml"
5         xmlns:h="http://xmlns.jcp.org/jsf/html">
6     <h:head>
7       <title>Display Current Time</title>
8       <meta http-equiv="refresh" content ="60" />
9     </h:head>
10    <h:body>
11      The current time is #{timeBean.time}
12    </h:body>
13  </html>
```

FIGURE 39.10 A New JSF page CurrentTime was created.

Line 11 defines a meta tag inside the h:head tag to tell the browser to refresh every 60 seconds. This line can also be written as

```
<meta http-equiv="refresh" content ="60"></ meta>
```

An element is called an *empty element* if there are no contents between the start tag and the end tag. In an empty element, data are typically specified as attributes in the start tag. You can close an empty element by placing a slash immediately preceding the start tag's right angle bracket, as shown in line 8, for brevity.

empty element

Line 8 uses a JSF expression `#{timeBean.time}` to obtain the current time. `timeBean` is an object of the `TimeBean` class. The object name can be changed in the `@Named` annotation (line 6 in Listing 39.2) using the following syntax:

```
@Named(name = "anyObjectName")
```

By default, the object name is the class name with the first letter in lowercase.

Note that `time` is a JavaBeans property because the `getTime()` method is defined in TimeBeans. The JSF expression can either use the property name or invoke the method to obtain the current time. So the following two expressions are fine:

```
#{timeBean.time}
#{timeBean.getTime()}
```

The syntax of a JSF expression is

```
#{expression}
```

JSF expressions bind JavaBeans objects with facelets. You will see more use of JSF expressions in the upcoming examples in this chapter.

Check Point

39.2.1 What is JSF?

39.2.2 How do you create a JSF project in NetBeans?

39.2.3 How do you create a JSF page in a JSF project?

39.2.4 What is a facelet?

39.2.5 What is the file extension name for a facelet?

39.2.6 What is a managed bean?

39.2.7 What is the `@Named` annotation for?

39.2.8 What is the `@RequestScope` annotation for?

39.3 JSF GUI Components

Key Point

JSF provides many elements for displaying GUI components.

Table 39.1 lists some of the commonly used elements. The tags with the `h` prefix are in the JSF HTML Tag library. The tags with the `f` prefix are in the JSF Core Tag library.

Listing 39.4 is an example that uses some of these elements to display a student registration form, as shown in Figure 39.11.

LISTING 39.4 StudentRegistrationForm.xhtml

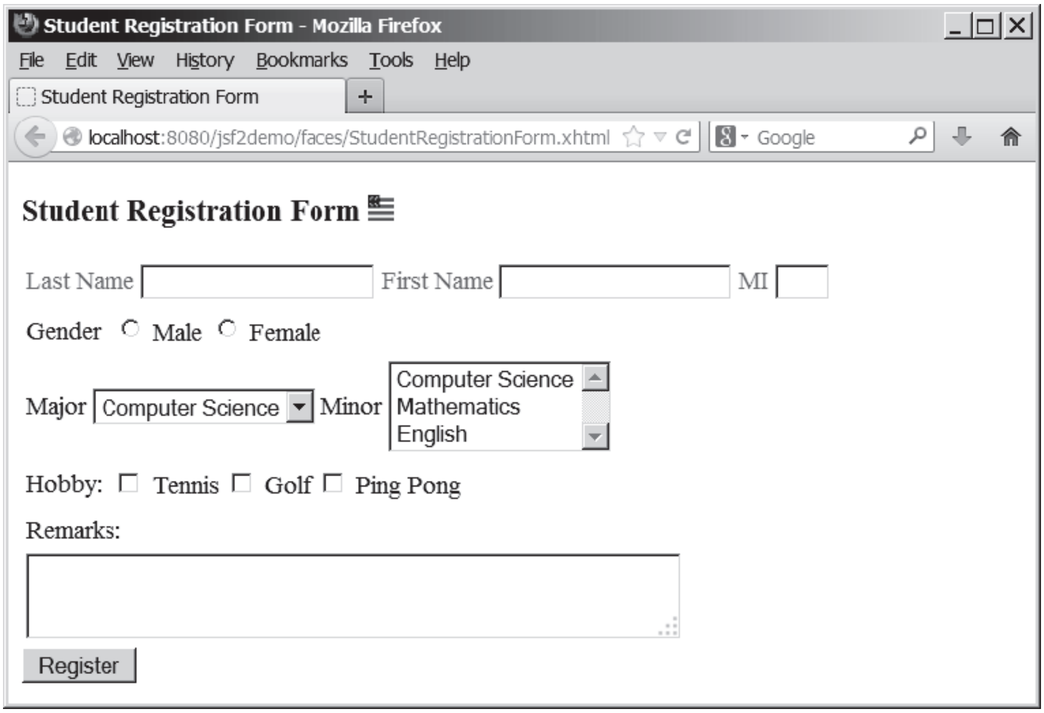
jsf core namespace

graphicImage

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
  <h:head>
    <title>Student Registration Form</title>
  </h:head>
  <h:body>
    <h:form>
      <!-- Use h:graphicImage -->
      <h3>Student Registration Form
         <h:graphicImage name="usIcon.gif" library="image"/>
      </h3>
```

TABLE 39.1 JSF GUI Form Elements

JSF Tag	*Description*
`h:form`	inserts an XHTML form into a page.
`h:panelGroup`	similar to a JavaFX FlowPane.
`h:panelGrid`	similar to a JavaFX GridPane.
`h:inputText`	displays a textbox for entering input.
`h:outputText`	displays a textbox for displaying output.
`h:inputTextArea`	displays a textarea for entering input.
`h:inputSecret`	displays a textbox for entering password.
`h:outputLabel`	displays a label.
`h:outputLink`	displays a hypertext link.
`h:selectOneMenu`	displays a combo box for selecting one item.
`h:selectOneRadio`	displays a set of radio button.
`h:selectManyCheckbox`	displays checkboxes.
`h:selectOneListbox`	displays a list for selecting one item.
`h:selectManyListbox`	displays a list for selecting multiple items.
`f:selectItem`	specifies an item in an **`h:selectOneMenu`**, **`h:selectOneRadio`**, or **`h:selectManyListbox`**.
`h:message`	displays a message for validating input.
`h:dataTable`	displays a data table.
`h:column`	specifies a column in a data table.
`h:graphicImage`	displays an image.


Student Registration Form - Mozilla Firefox
File
Edit
View
History
Bookmarks
Tools
Help
Student Registration Form
localhost:8080/jsf2demo/faces/StudentRegistrationForm.xhtml
Google
Register

Student Registration Form

Last Name First Name MI

Gender ○ Male ○ Female

Major Computer Science Minor Computer Science Mathematics English

Hobby: □ Tennis □ Golf □ Ping Pong

Remarks:

FIGURE 39.11 A student registration form is displayed using JSF elements.

h:panelGrid
h:outputLabel
h:inputText
h:selectOneRadio
f:selectItem
h:selectOneMenu
h:selectManyListBox
h:selectManyCheckbox
h:inputTextarea
h:commandButton

```

      <!-- Use h:panelGrid -->
      <h:panelGrid columns="6" style="color:green">
        <h:outputLabel value="Last Name"/>
        <h:inputText id="lastNameInputText" />
        <h:outputLabel value="First Name" />
        <h:inputText id="firstNameInputText" />
        <h:outputLabel value="MI" />
        <h:inputText id="miInputText" size="1" />
      </h:panelGrid>

      <!-- Use radio buttons -->
      <h:panelGrid columns="2">
        <h:outputLabel>Gender </h:outputLabel>
        <h:selectOneRadio id="genderSelectOneRadio">
          <f:selectItem itemValue="Male"
                        itemLabel="Male"/>
          <f:selectItem itemValue="Female"
                        itemLabel="Female"/>
        </h:selectOneRadio>
      </h:panelGrid>

      <!-- Use combo box and list -->
      <h:panelGrid columns="4">
        <h:outputLabel value="Major "/>
        <h:selectOneMenu id="majorSelectOneMenu">
          <f:selectItem itemValue="Computer Science"/>
          <f:selectItem itemValue="Mathematics"/>
        </h:selectOneMenu>
        <h:outputLabel value="Minor "/>
        <h:selectManyListbox id="minorSelectManyListbox">
          <f:selectItem itemValue="Computer Science"/>
          <f:selectItem itemValue="Mathematics"/>
          <f:selectItem itemValue="English"/>
        </h:selectManyListbox>
      </h:panelGrid>

      <!-- Use check boxes -->
      <h:panelGrid columns="4">
        <h:outputLabel value="Hobby: "/>
        <h:selectManyCheckbox id="hobbySelectManyCheckbox">
          <f:selectItem itemValue="Tennis"/>
          <f:selectItem itemValue="Golf"/>
          <f:selectItem itemValue="Ping Pong"/>
        </h:selectManyCheckbox>
      </h:panelGrid>

      <!-- Use text area -->
      <h:panelGrid columns="1">
        <h:outputLabel>Remarks:</h:outputLabel>
        <h:inputTextarea id="remarksInputTextarea"
                         style="width:400px; height:50px;" />
      </h:panelGrid>

      <!-- Use command button -->
      <h:commandButton value="Register" />
    </h:form>
  </h:body>
</html>
```

jsf core xmlns

The tags with prefix `f` are in the JSF core tag library. Line 6

```
xmlns:f="http://xmlns.jcp.org/jsf/core">
```

locates the library for these tags.

h:graphicImage

The `h:graphicImage` tag displays an image in the file usIcon.gif (line 14). The file is located in the /resources/image folder. In JSF 2.0, all resources (image files, audio files, and CCS files) should be placed under the `resources` folder under the `Web Pages` node. You can create these folders as follows:

Step 1: Right-click the `Web Pages` node in the project pane to display a context menu and choose *New*, *Folder* to display the New Folder dialog box. (If *Folder* is not in the context menu, choose *Other* to locate it.)

Step 2: Enter `resources` as the Folder Name and click *Finish* to create the `resources` folder, as shown in Figure 39.12.

Step 3: Right-click the `resources` node in the project pane to create the image folder under resources. You can now place usIcon.gif under the image folder.

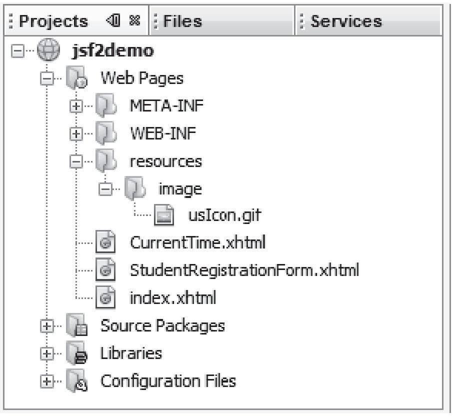


Figure 39.12 The resources folder was created.

h:panelGrid

JSF provides `h:panelGrid` and `h:panelGroup` elements to contain and layout subelements. `h:panelGrid` places the elements in a grid like the JavaFX `GridPane`. `h:panelGrid` places the elements in a grid with the specified number of columns. Lines 18–25 place six elements (labels and input texts) that are in an `h:panelGrid`. The `columns` attribute specifies that each row in the grid has `6` columns. The elements are placed into a row from left to right in the order they appear in the facelet. When a row is full, a new row is created to hold the elements. We used `h:panelGrid` in this example. You may replace it with `h:panelGroup` to see how the elements would be arranged.

the `style` attribute

You may use the `style` attribute with a JSF html tag to specify the CSS style for the element and its subelements. The `style` attribute in line 18 specifies the color green for all elements in this `h:panelGrid` element.

h:outputLabel

The `h:outputLabel` element is for displaying a label (line 19). The `value` attribute specifies the label's text.

h:inputText

The `h:inputText` element is for displaying a text input box for the user to enter a text (line 20). The `id` attribute is useful for other elements or the server program to reference this element.

h:selectOneRadio

The `h:selectOneRadio` element is for displaying a group of radio buttons (line 30). Each radio button is defined using an `f:selectItem` element (lines 31–34).

h:selectOneMenu

The h:selectOneMenu element is for displaying a combo box (line 41). Each item in the combo box is defined using an f:selectItem element (lines 42 and 43).

h:selectManyListbox

The h:selectManyListbox element is for displaying a list for the user to choose multiple items in a list (line 46). Each item in the list is defined using an f:selectItem element (lines 47–49).

h:selectManyCheckbox

The h:selectManyCheckbox element is for displaying a group of check boxes (line 56). Each item in the check box is defined using an f:selectItem element (lines 57–59).

h:selectTextarea

The h:selectTextarea element is for displaying a text area for multiple lines of input (line 66). The style attribute is used to specify the width and height of the text area (line 67).

h:commandButton

The h:commandButton element is for displaying a button (line 71). When the button is clicked, an action is performed. The default action is to request the same page from the server. The next section shows how to process the form.

Check Point

39.3.1 What is the name space for JSF tags with prefix h and prefix f?

39.3.2 Describe the use of the following tags?

```
h:form, h:panelGroup, h:panelGrid, h:inputText, h:outputText,
h:inputTextArea, h:inputSecret, h:outputLabel, h:outputLink,
h:selectOneMenu, h:selectOneRadio, h:selectBooleanCheckbox,
h:selectOneListbox, h:selectManyListbox, h:selectItem,
h:message, h:dataTable, h:columm, h:graphicImage
```

Key Point

39.4 Processing the Form

Processing forms is a common task for Web programming. JSF provides tools for processing forms.

The preceding section introduced how to display a form using common JSF elements. This section shows how to obtain and process the input.

To obtain input from the form, simply bind each input element with a property in a managed bean. We now define a managed bean named registration as shown in Listing 39.5.

Listing 39.5 RegistrationJSFBean.java

managed bean
request scope
property lastName

```
package jsf2demo;

import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@Named(value = "registration")
@RequestScoped
public class RegistrationJSFBean {
  private String lastName;
  private String firstName;
  private String mi;
  private String gender;
  private String major;
  private String[] minor;
  private String[] hobby;
  private String remarks;

  public RegistrationJSFBean() {
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getMi() {
    return mi;
  }

  public void setMi(String mi) {
    this.mi = mi;
  }

  public String getGender() {
    return gender;
  }

  public void setGender(String gender) {
    this.gender = gender;
  }

  public String getMajor() {
    return major;
  }

  public void setMajor(String major) {
    this.major = major;
  }

  public String[] getMinor() {
    return minor;
  }

  public void setMinor(String[] minor) {
    this.minor = minor;
  }

  public String[] getHobby() {
    return hobby;
  }

  public void setHobby(String[] hobby) {
    this.hobby = hobby;
  }

  public String getRemarks() {
    return remarks;
  }

  public void setRemarks(String remarks) {
    this.remarks = remarks;
  }
```

getResponse()

```

  public String getResponse() {
    if (lastName == null)
      return ""; // Request has not been made
    else {
      String allMinor = "";
      for (String s: minor) {
        allMinor += s + " ";
      }

      String allHobby = "";
      for (String s: hobby) {
        allHobby += s + " ";
      }

      return "<p style=\"color:red\">You entered <br />" +
        "Last Name: " + lastName + "<br />" +
        "First Name: " + firstName + "<br />" +
        "MI: " + mi + "<br />" +
        "Gender: " + gender + "<br />" +
        "Major: " + major + "<br />" +
        "Minor: " + allMinor + "<br />" +
        "Hobby: " + allHobby + "<br />" +
        "Remarks: " + remarks + "</p>";
    }
  }
}
```

bean properties

The `RegistrationJSFBean` class is a managed bean that defines the properties `lastName`, `firstName`, `mi`, `gender`, `major`, `minor`, and `remarks`, which will be bound to the elements in the JSF registration form.

The registration form can now be revised as shown in Listing 39.6. Figure 39.13 shows that new JSF page displays the user input upon clicking the *Register* button.

Listing 39.6 ProcessStudentRegistrationForm.xhtml

jsf core namespace

bind lastName

bind firstName

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
  <h:head>
    <title>Student Registration Form</title>
  </h:head>
  <h:body>
    <h:form>
      <!-- Use h:graphicImage -->
      <h3>Student Registration Form
        <h:graphicImage name="usIcon.gif" library="image"/>
      </h3>

      <!-- Use h:panelGrid -->
      <h:panelGrid columns="6" style="color:green">
        <h:outputLabel value="Last Name"/>
        <h:inputText id="lastNameInputText"
                     value="#{registration.lastName}"/>
        <h:outputLabel value="First Name" />
        <h:inputText id="firstNameInputText"
                     value="#{registration.firstName}"/>
        <h:outputLabel value="MI" />
        <h:inputText id="miInputText" size="1"
                     value="#{registration.mi}"/>
        </h:panelGrid>

        <!-- Use radio buttons -->
        <h:panelGrid columns="2">
          <h:outputLabel>Gender </h:outputLabel>
          <h:selectOneRadio id="genderSelectOneRadio"
                            value="#{registration.gender}">
          <f:selectItem itemValue="Male"
                        itemLabel="Male"/>
          <f:selectItem itemValue="Female"
                        itemLabel="Female"/>
        </h:selectOneRadio>
      </h:panelGrid>

      <!-- Use combo box and list -->
      <h:panelGrid columns="4">
        <h:outputLabel value="Major "/>
        <h:selectOneMenu id="majorSelectOneMenu"
                         value="#{registration.major}">
          <f:selectItem itemValue="Computer Science"/>
          <f:selectItem itemValue="Mathematics"/>
        </h:selectOneMenu>
        <h:outputLabel value="Minor "/>
        <h:selectManyListbox id="minorSelectManyListbox"
                             value="#{registration.minor}">
          <f:selectItem itemValue="Computer Science"/>
          <f:selectItem itemValue="Mathematics"/>
          <f:selectItem itemValue="English"/>
        </h:selectManyListbox>
      </h:panelGrid>

      <!-- Use check boxes -->
      <h:panelGrid columns="4">
        <h:outputLabel value="Hobby: "/>
        <h:selectManyCheckbox id="hobbySelectManyCheckbox"
                              value="#{registration.hobby}">
          <f:selectItem itemValue="Tennis"/>
          <f:selectItem itemValue="Golf"/>
          <f:selectItem itemValue="Ping Pong"/>
        </h:selectManyCheckbox>
      </h:panelGrid>

      <!-- Use text area -->
      <h:panelGrid columns="1">
        <h:outputLabel>Remarks:</h:outputLabel>
        <h:inputTextarea id="remarksInputTextarea"
                         style="width:400px; height:50px;"
                         value="#{registration.remarks}"/>
      </h:panelGrid>

      <!-- Use command button -->
      <h:commandButton value="Register" />
      <br />
      <h:outputText escape="false" style="color:red"
                    value="#{registration.response}" />
    </h:form>
  </h:body>
</html>
```

bind mi

bind gender

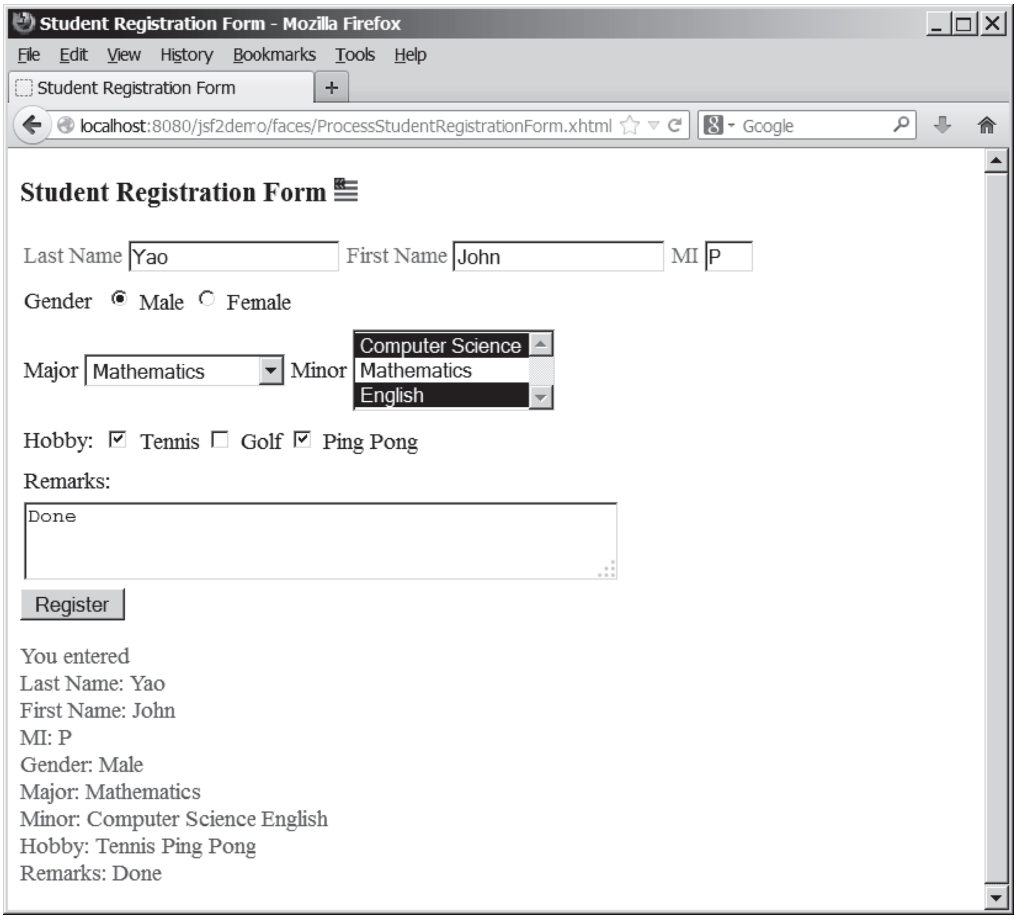
bind major

bind minor

bind hobby

bind remarks

bind response


Student Registration Form - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Student Registration Form
localhost:8080/jsf2demo/faces/ProcessStudentRegistrationForm.xhtml
Google
Register

Student Registration Form

Last Name Yao First Name John MI P

Gender ● Male ○ Female

Major Mathematics Minor Computer Science
Mathematics
English

Hobby: ☑ Tennis □ Golf ☑ Ping Pong

Remarks:

```
Done
```

You entered
Last Name: Yao
First Name: John
MI: P
Gender: Male
Major: Mathematics
Minor: Computer Science English
Hobby: Tennis Ping Pong
Remarks: Done

FIGURE 39.13 The user input is collected and displayed after clicking the *Register* button.

binding input texts

The new JSF form in this listing binds the `h:inputText` element for last name, first name, and mi with the properties `lastName`, `firstName`, and `mi` in the managed bean (lines 21, 24, and 27). When the *Register* button is clicked, the page is sent to the server, which invokes the setter methods to set the properties in the managed bean.

binding radio buttons

The `h:selectOneRadio` element is bound to the `gender` property (line 34). Each radio button has an `itemValue`. The selected radio button's `itemValue` is set to the `gender` property in the bean when the page is sent to the server.

binding combo box

The `h:selectOneMenu` element is bound to the `major` property (line 46). When the page is sent to the server, the selected item is returned as a string and is set to the `major` property.

binding list box

The `h:selectManyListbox` element is bound to the `minor` property (line 52). When the page is sent to the server, the selected items are returned as an array of strings and set to the `minor` property.

binding check boxes

The `h:selectManyCheckbox` element is bound to the `hobby` property (line 63). When the page is sent to the server, the checked boxes are returned as an array of `itemValues` and set to the `hobby` property.

binding text area

The `h:selectTextarea` element is bound to the `remarks` property (line 75). When the page is sent to the server, the content in the text area is returned as a string and set to the `remarks` property.

binding response

The `h:outputText` element is bound to the `response` property (line 82). This is a read-only property in the bean. It is `""` if `lastName` is `null` (lines 86 and 87 in Listing 39.5). When the page is returned to the client, the `response` property value is displayed in the output text element (line 82).

The `h:outputText` element's `escape` attribute is set to `false` (line 81) to enable the contents to be displayed in HTML. By default, the `escape` attribute is `true`, which indicates the contents are considered regular text.

escape attribute

39.4.1 In the `h:outputText` tag, what is the `escape` attribute for?

39.4.2 Does every GUI component tag in JSF have the style attribute?

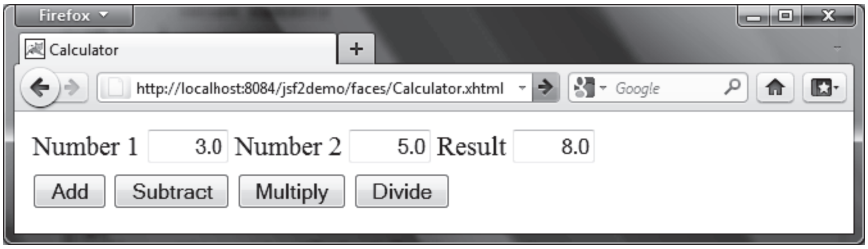
Check Point

39.5 Case Study: Calculator

This section gives a case study on using GUI elements and processing forms.

Key Point

This section uses JSF to develop a calculator to perform addition, subtraction, multiplication, and division, as shown in Figure 39.14.



Number 1 3.0 Number 2 5.0 Result 8.0

FIGURE 39.14 This JSF application enables you to perform addition, subtraction, multiplication, and division.

Here are the steps to develop this project:

Step 1. Create a new managed bean named `calculator` with the request scope as shown in Listing 39.7, CalculatorJSFBean.java.

create managed bean

Step 2. Create a JSF facelet in Listing 39.8, Calculator.xhtml.

create JSF facelet

LISTING 39.7 `CalculatorJSFBean.java`

```
package jsf2demo;

import javax.inject.Named;
import javax.enterprise.context.RequestScoped;

@Named(value = "calculator")
@RequestScoped
public class CalculatorJSFBean {
  private Double number1;
  private Double number2;
  private Double result;

  public CalculatorJSFBean() {
  }

  public Double getNumber1() {
    return number1;
  }

  public Double getNumber2() {
    return number2;
  }

```

property number1
property number2
property result

```
24    public Double getResult() {
25      return result;
26    }
27
28    public void setNumber1(Double number1) {
29      this.number1 = number1;
30    }
31
32    public void setNumber2(Double number2) {
33      this.number2 = number2;
34    }
35
36    public void setResult(Double result) {
37      this.result = result;
38    }
39
40    public void add() {
41      result = number1 + number2;
42    }
43
44    public void subtract() {
45      result = number1 - number2;
46    }
47
48    public void divide() {
49      result = number1 / number2;
50    }
51
52    public void multiply() {
53      result = number1 * number2;
54    }
55  }
```

add

subtract

divide

multiply

The managed bean has three properties **number1**, **number2**, and **result** (lines 9–38). The methods **add()**, **subtract()**, **divide()**, and **multiply()** add, subtract, multiply, and divide **number1** with **number2** and assigns the result to **result** (lines 40–54).

LISTING 39.8 Calculator.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
  <h:head>
    <title>Calculator</title>
  </h:head>
  <h:body>
    <h:form>
      <h:panelGrid columns="6">
        <h:outputLabel value="Number 1"/>
        <h:inputText id="number1InputText" size ="4"
                     style="text-align: right"
                     value="#{calculator.number1}"/>
        <h:outputLabel value="Number 2" />
        <h:inputText id="number2InputText" size ="4"
                     style="text-align: right"
                     value="#{calculator.number2}"/>
        <h:outputLabel value="Result" />
        <h:inputText id="resultInputText" size ="4"
```

right align

bind text input

```
22                       style="text-align: right"
23                       value="#{calculator.result}"/>
24        </h:panelGrid>
25
26        <h:panelGrid columns="4">
27          <h:commandButton value="Add"
28                           action ="#{calculator.add}"/>
29          <h:commandButton value="Subtract"
30                           action ="#{calculator.subtract}"/>
31          <h:commandButton value="Multiply"
32                           action ="#{calculator.multiply}"/>
33          <h:commandButton value="Divide"
34                           action ="#{calculator.divide}"/>
35        </h:panelGrid>
36      </h:form>
37    </h:body>
38  </html>
```

action

Three text input components along with their labels are placed in the grid panel (lines 11–24). Four button components are placed in the grid panel (lines 26–35).

The bean property **number1** is bound to the text input for Number 1 (line 15). The CSS style **text-align: right** (line 14) specifies that the text is right aligned in the input box.

The **action** attribute for the *Add* button is set to the **add** method in the calculator bean (line 28). When the *Add* button is clicked, the **add** method in the bean is invoked to add **number1** with **number2** and assign the result to **result**. Since the **result** property is bound to the Result input text (line 23), the new result is now displayed in the text input field.

39.6 Session Tracking

You can create a managed bean at the application scope, session scope, view scope, or request scope.

Key Point

JSF supports session tracking for managed beans at the application scope, session scope, view scope, and request scope. The *scope* is the lifetime of a bean. A *request*-scoped bean is alive in a single HTTP request. After the request is processed, the bean is no longer alive. A *view*-scoped bean lives as long as you are in the same JSF page. A *session*-scoped bean is alive for the entire Web session between a client and the server. An *application*-scoped bean lives as long as the Web application runs. In essence, a request-scoped bean is created once for a request; a view-scoped bean is created once for the view; a session-scoped bean is created once for the entire session; and an application-scoped bean is created once for the entire application. A managed bean with a session scope must be serializable because the system may need to free resources during and session and stores the bean to a file if the bean is not used for a while. When the bean is used again, the system will restore the bean to the memory.

scope
request scope
view scope
session scope
application scope

Consider the following example that prompts the user to guess a number. When the page starts, the program randomly generates a number between **0** and **99**. This number is stored in a bean. When the user enters a guess, the program checks the guess with the random number in the bean and tells the user whether the guess is too high, too low, or just right, as shown in Figure 39.15.

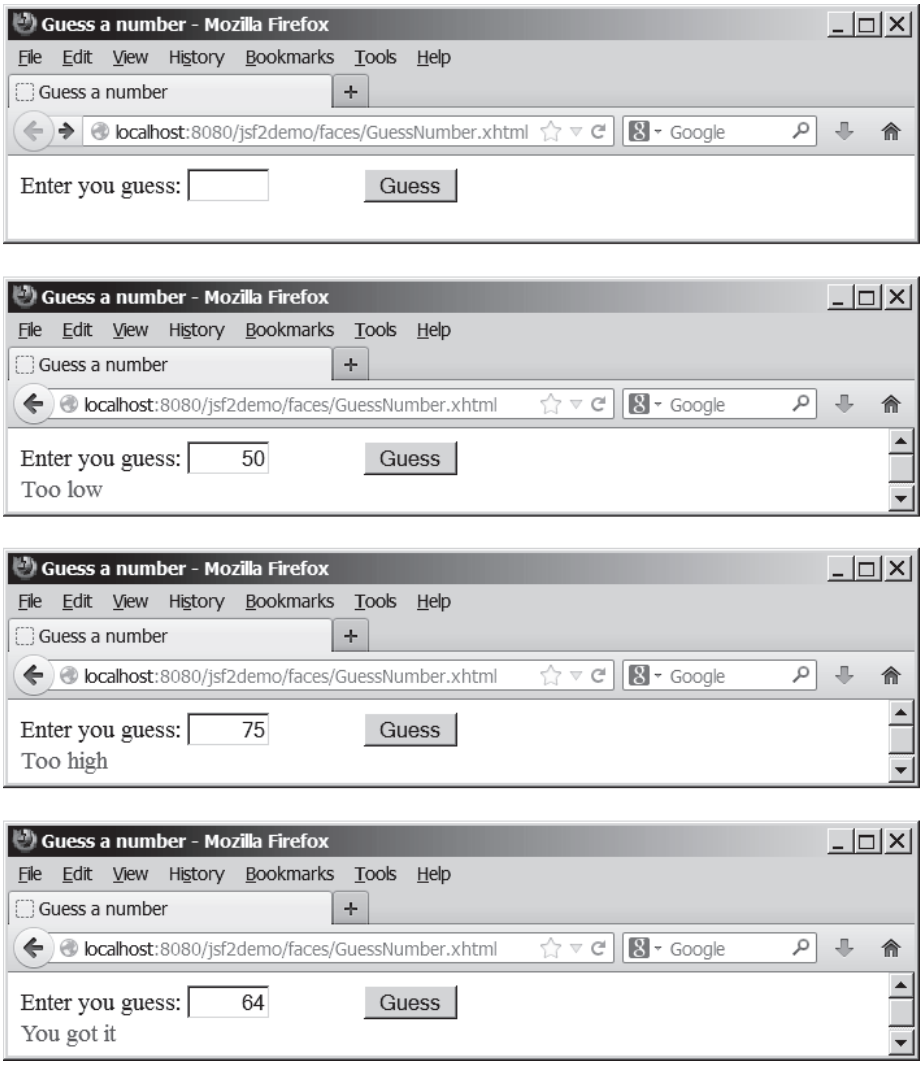
Here are the steps to develop this project:

Step 1. Create a new managed bean named **guessNumber** with the view scope as shown in Listing 39.9, GuessNumberJSFBean.java.

create managed bean

Step 2. Create a JSF facelet in Listing 39.10, GuessNumber.xhtml.

create JSF facelet


Guess a number - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Guess a number
localhost:8080/jsf2demo/faces/GuessNumber.xhtml
Google
Guess

Enter you guess:

Enter you guess: 50
Too low

Enter you guess: 75
Too high

Enter you guess: 64
You got it

FIGURE 39.15 The user enters a guess and the program displays the result.

LISTING 39.9 GuessNumberJSFBean.java

view scope

random number

guess by user

create random number

getter method

```
package jsf2demo;

import javax.inject.Named;
import javax.faces.view.ViewScoped;

@Named(value = "guessNumber")
@ViewScoped
public class GuessNumberJSFBean {
  private int number;
  private String guessString;

  public GuessNumberJSFBean() {
   number = (int)(Math.random() * 100);
  }

  public String getGuessString() {
    return guessString;
  }
19
20    public void setGuessString(String guessString) {
21      this.guessString = guessString;
22    }
23
24    public String getResponse() {
25      if (guessString == null)
26        return ""; // No user input yet
27
28      int guess = Integer.parseInt(guessString);
29      if (guess < number)
30        return "Too low";
31      else if (guess == number)
32        return "You got it";
33      else
34        return "Too high";
35    }
36  }
```

setter method

get response

check guess

The managed bean uses the `@ViewScope` annotation (line 7) to set up the view scope for the bean. The view scope is most appropriate for this project. The bean is alive as long as the view is not changed. The bean is created when the page is displayed for the first time. A random number between `0` and `99` is assigned to `number` (line 13) when the bean is created. This number will not change as long as the bean is alive in the same view.

The `getResponse` method converts `guessString` from the user input to an integer (line 28) and determines if the guess is too low (line 30), too high (line 34), and just right (line 32).

LISTING 39.10 GuessNumber.xhtml

```
 1  <?xml version='1.0' encoding='UTF-8' ?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
 4  <html xmlns="http://www.w3.org/1999/xhtml"
 5        xmlns:h="http://xmlns.jcp.org/jsf/html">
 6    <h:head>
 7      <title>Guess a number</title>
 8    </h:head>
 9    <h:body>
10      <h:form>
11        <h:outputLabel value="Enter you guess: "/>
12        <h:inputText style="text-align: right; width: 50px"
13                     id="guessInputText"
14                     value="#{guessNumber.guessString}"/>
15        <h:commandButton style="margin-left: 60px" value="Guess" />
16        <br />
17        <h:outputText style="color: red"
18                      value="#{guessNumber.response}" />
19      </h:form>
20    </h:body>
21  </html>
```

bind text input

bind text output

The bean property `guessString` is bound to the text input (line 14). The CSS style `text-align: right` (line 13) specifies that the text is right aligned in the input box.

The CSS style `margin-left: 60px` (line 15) specifies that the command button has a left margin of 60 pixels.

The bean property `response` is bound to the text output (line 18). The CSS style `color: red` (line 17) specifies that the text is displayed in red in the output box.

scope

The project uses the **view** scope. What happens if the scope is changed to the **request** scope? Every time the page is refreshed, JSF creates a new bean with a new random number. What happens if the scope is changed to the **session** scope? The bean will be alive as long as the browser is alive. What happens if the scope is changed to the **application** scope? The bean will be created once when the application is launched from the server. So every client will use the same random number.

Check Point

39.6.1 What is a scope? What are the available scopes in JSF? Explain request scope, view scope, session scope, and application scope. How do you set a request scope, view scope, session scope, and application scope in a managed bean?

39.6.2 What happens if the bean scope in Listing 39.9, GuessNumberJSFBean.java is changed to request?

39.6.3 What happens if the bean scope in Listing 39.9, GuessNumberJSFBean.java is changed to session?

39.6.4 What happens if the bean scope in Listing 39.9, GuessNumberJSFBean.java is changed to application?

39.7 Validating Input

Key Point

JSF provides tools for validating user input.

In the preceding `GuessNumber` page, an error would occur if you entered a noninteger in the input box before clicking the *Guess* button. One way to fix the problem is to check the text field before processing any event. But a better way is to user the validators. You can use the standard validator tags in the JSF Core Tag Library or create custom validators. Table 39.2 lists some JSF input validator tags.

TABLE 39.2 JSF Input Validator Tags

JSF Tag	*Description*
`f:validateLength`	validates the length of the input.
`f:validateDoubleRange`	validates whether numeric input falls within acceptable range of double values.
`f:validateLongRange`	validates whether numeric input falls within acceptable range of long values.
`f:validateRequired`	validates whether a field is not empty.
`f:validateRegex`	validates whether the input matches a regualar expression.
`f:validateBean`	invokes a custom method in a bean to perform custom validation.

Consider the following example that displays a form for collecting user input as shown in Figure 39.16. All text fields in the form must be filled. If not, error messages are displayed. The SSN must be formatted correctly. If not, an error is displayed. If all input are correct, clicking *Submit* displays the result in an output text, as shown in Figure 39.17.

Here are the steps to create this project.

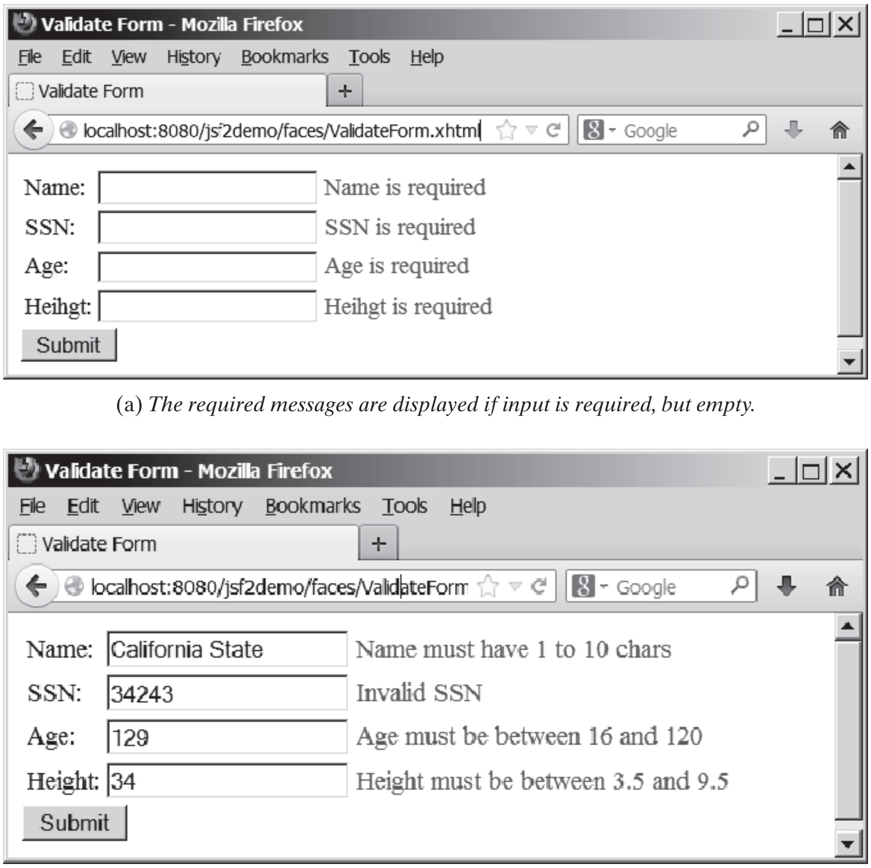
Step 1. Create a new page in Listing 39.11, ValidateForm.xhtml.

Step 2. Create a new managed bean named `validateForm`, as shown in Listing 39.12.

LISTING 39.11 ValidateForm.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
```

```
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://xmlns.jcp.org/jsf/core">
<h:head>
```


Validate Form - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Validate Form
localhost:8080/jsf2demo/faces/ValidateForm.xhtml
Google
Submit

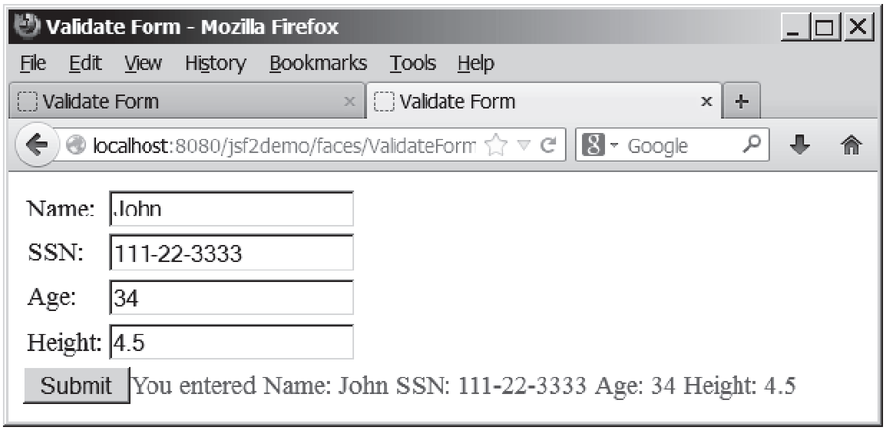
Name: Name is required
SSN: SSN is required
Age: Age is required
Heihgt: Heihgt is required

(a) *The required messages are displayed if input is required, but empty.*

Name: California State Name must have 1 to 10 chars
SSN: 34243 Invalid SSN
Age: 129 Age must be between 16 and 120
Height: 34 Height must be between 3.5 and 9.5

(b) *Error messages are displayed if input is incorrect.*

FIGURE 39.16 The input fields are validated.


Validate Form - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Validate Form
Validate Form
localhost:8080/jsf2demo/faces/ValidateForm
Google
Submit

Name: John
SSN: 111-22-3333
Age: 34
Height: 4.5
You entered Name: John SSN: 111-22-3333 Age: 34 Height: 4.5

FIGURE 39.17 The correct input values are displayed.

required input
required message
validator message
validate length
message element
validate regex
validate integer range
validate double range

```
 8      <title>Validate Form</title>
 9    </h:head>
10    <h:body>
11      <h:form>
12        <h:panelGrid columns="3">
13          <h:outputLabel value="Name:"/>
14          <h:inputText id="nameInputText" required="true"
15            requiredMessage="Name is required"
16            validatorMessage="Name must have 1 to 10 chars"
17            value="#{validateForm.name}">
18            <f:validateLength minimum="1" maximum="10" />
19          </h:inputText>
20          <h:message for="nameInputText" style="color:red"/>
21
22          <h:outputLabel value="SSN:" />
23          <h:inputText id="ssnInputText" required="true"
24            requiredMessage="SSN is required"
25            validatorMessage="Invalid SSN"
26            value="#{validateForm.ssn}">
27            <f:validateRegex pattern="[\d]{3}-[\d]{2}-[\d]{4}"/>
28          </h:inputText>
29          <h:message for="ssnInputText" style="color:red"/>
30
31          <h:outputLabel value="Age:" />
32          <h:inputText id="ageInputText" required="true"
33            requiredMessage="Age is required"
34            validatorMessage="Age must be between 16 and 120"
35            value="#{validateForm.ageString}">
36            <f:validateLongRange minimum="16" maximum="120"/>
37          </h:inputText>
38          <h:message for="ageInputText" style="color:red"/>
39
40          <h:outputLabel value="Height:" />
41          <h:inputText id="heightInputText" required="true"
42            requiredMessage="Height is required"
43            validatorMessage="Height must be between 3.5 and 9.5"
44            value="#{validateForm.heightString}">
45            <f:validateDoubleRange minimum="3.5" maximum="9.5"/>
46          </h:inputText>
47          <h:message for="heightInputText" style="color:red"/>
48        </h:panelGrid>
49
50        <h:commandButton value="Submit" />
51
52        <h:outputText style="color:red"
53                      value="#{validateForm.response}" />
54      </h:form>
55    </h:body>
56  </html>
```

required attribute
requiredMessage

For each input text field, set its `required` attribute `true` (lines 14, 23, 32, and 41) to indicate that an input value is required for the field. When a required input field is empty, the `requiredMessage` is displayed (lines 15, 24, 33, and 42).

validatorMessage
f:validateLength

The `validatorMessage` attribute specifies a message to be displayed if the input field is invalid (line 16). The `f:validateLength` tag specifies the minimum or maximum length of the input (line 18). JSF will determine whether the input length is valid.

h:message

The `h:message` element displays the `validatorMessage` if the input is invalid. The element's `for` attribute specifies the `id` of the element for which the message will be displayed (line 20).

f:validateRegex

The `f:validateRegex` tag specifies a regular expression for validating the input (line 27). For information on regular expression, see Appendix H.

The f:validateLongRange tag specifies a range for an integer input using the minimum and maximum attributes (line 45). In this project, a valid age value is between 16 and 120.

f:validateLongRange

The f:validateDoubleRange tag specifies a range for a double input using the minimum and maximum attributes (line 36). In this project, a valid height value is between 3.5 and 9.5.

f:validateDoubleRange

Listing 39.12 ValidateFormJSFBean.java

```java
package jsf2demo;

import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@Named(value = "validateForm")
@RequestScoped
public class ValidateFormJSFBean {
  private String name;
  private String ssn;
  private String ageString;
  private String heightString;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getSsn() {
    return ssn;
  }

  public void setSsn(String ssn) {
    this.ssn = ssn;
  }

  public String getAgeString() {
    return ageString;
  }

  public void setAgeString(String ageString) {
    this.ageString = ageString;
  }

  public String getHeightString() {
    return heightString;
  }

  public void setHeightString(String heightString) {
    this.heightString = heightString;
  }

  public String getResponse() {
    if (name == null || ssn == null || ageString == null
            || heightString == null) {
      return "";
    }
    else {
      return "You entered " +
        " Name: " + name +
        " SSN: " + ssn +
```

some input not set

```
55          " Age: " + ageString +
56          " Height: " + heightString;
57      }
58    }
59  }
```

If an input is invalid, its value is not set to the bean. So only when all input are correct, the `getResponse()` method will return all input values (lines 46–58).

39.7.1 Write a tag that validates an input text with minimal length of `2` and maximum `12`.

39.7.2 Write a tag that validates an input text for SSN using a regular expression.

39.7.3 Write a tag that validates an input text for a double value with minimal `4.5` and maximum `19.9`.

39.7.4 Write a tag that validates an input text for an integer value with minimal `4` and maximum `20`.

39.7.5 Write a tag that makes an input text required.

39.8 Binding Database with Facelets

You can bind a database in JSF applications.

Often you need to access a database from a webpage. This section gives examples of building Web applications using databases.

Consider the following example that lets the user choose a course, as shown in Figure 39.18. After a course is selected in the combo box, the students enrolled in the course are displayed in the table, as shown in Figure 39.19. In this example, all the course titles in the `Course` table are bound to the combo box and the query result for the students enrolled in the course is bound to the table.

Here are the steps to create this project:

managed bean

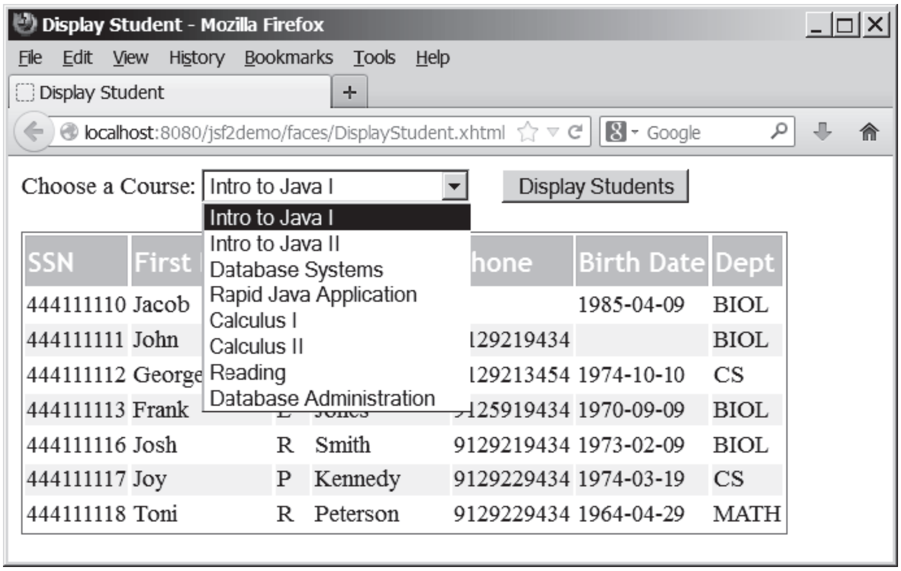
Step 1. Create a managed bean named `courseName` with application scope, as shown in Listing 39.13.

JSF page

Step 2. Create a JSF in Listing 39.14, DisplayStudent.xhtml.

style sheet

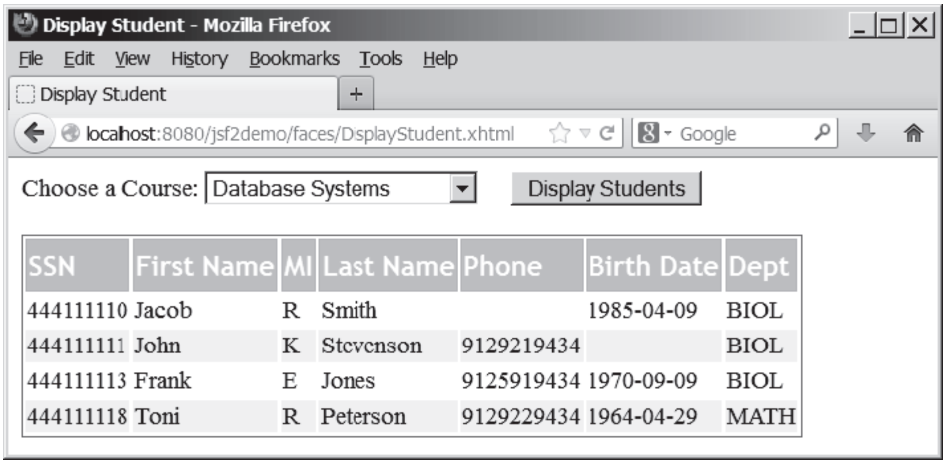
Step 3. Create a cascading style sheet for formatting the table as follows:



Choose a Course: Intro to Java I

SSN	First			hone	Birth Date	Dept
444111110	Jacob				1985-04-09	BIOL
444111111	John			129219434		BIOL
444111112	George			129213454	1974-10-10	CS
444111113	Frank			125919434	1970-09-09	BIOL
444111116	Josh	R	Smith	9129219434	1973-02-09	BIOL
444111117	Joy	P	Kennedy	9129229434	1974-03-19	CS
444111118	Toni	R	Peterson	9129229434	1964-04-29	MATH

FIGURE 39.18 You need to choose a course and display the students enrolled in the course.



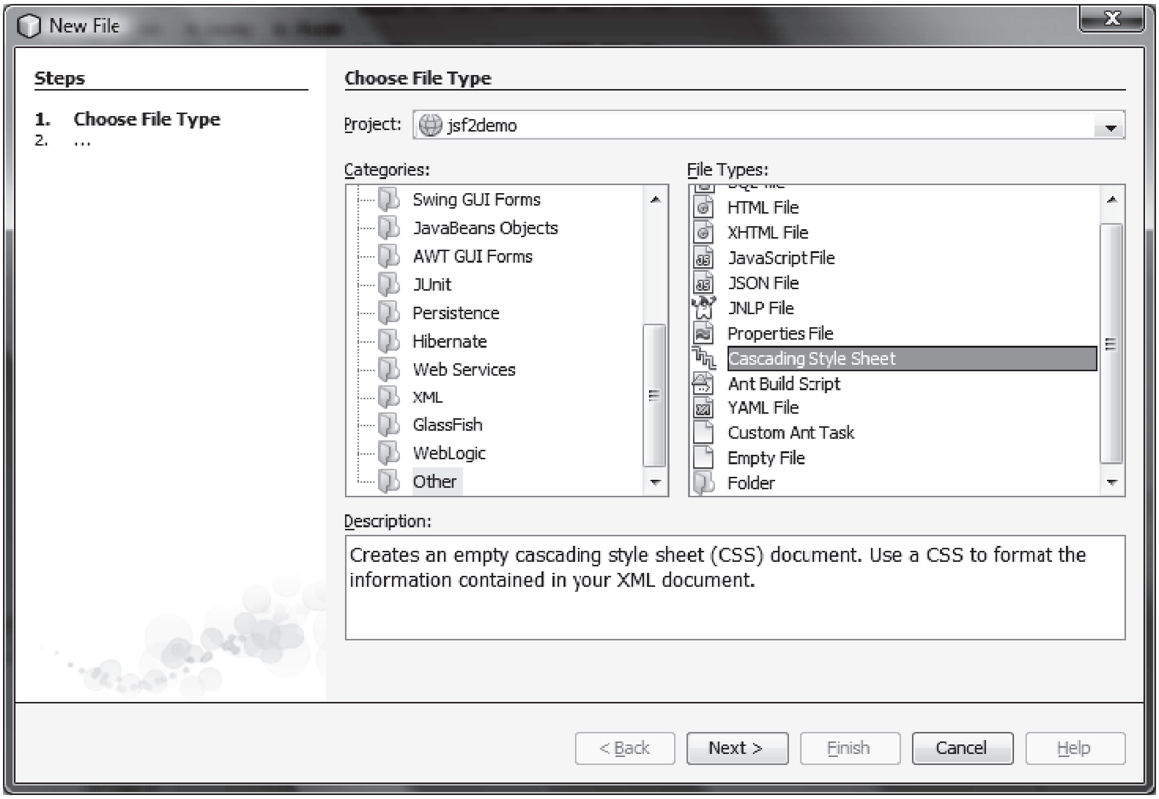
Display Student - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Display Student
localhost:8080/jsf2demo/faces/DsplayStudent.xhtml
Google

Choose a Course: Database Systems

Display Students

SSN	First Name	MI	Last Name	Phone	Birth Date	Dept
444111110	Jacob	R	Smith		1985-04-09	BIOL
444111111	John	K	Stevenson	9129219434		BIOL
444111113	Frank	E	Jones	9125919434	1970-09-09	BIOL
444111118	Toni	R	Peterson	9129229434	1964-04-29	MATH

FIGURE 39.19 The table displays the students enrolled in the course.



Project: jsf2demo

Categories:

File Types:

Description:

Creates an empty cascading style sheet (CSS) document. Use a CSS to format the information contained in your XML document.

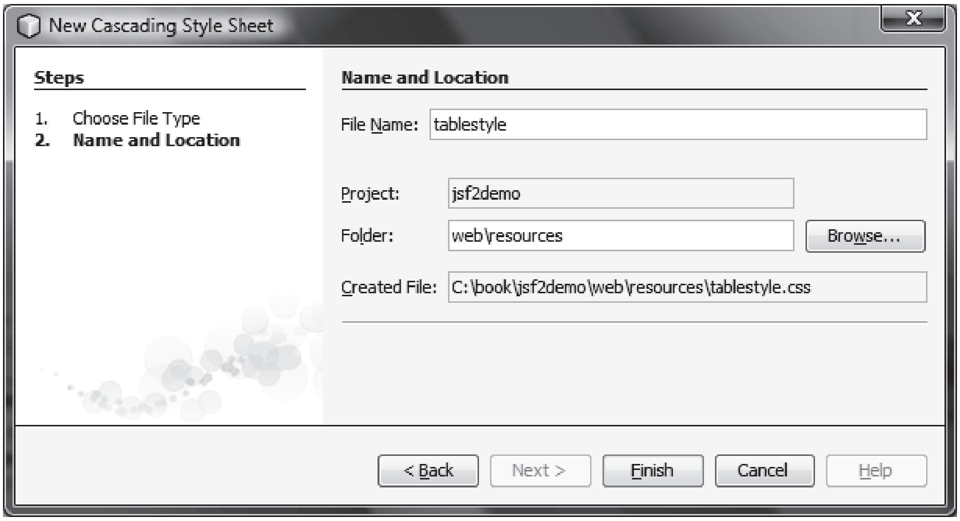
FIGURE 39.20 You can create CSS files for Web project in NetBenas.

Step 3.1. Right-click the `resources` node to choose *New*, *Others* to display the New File dialog box, as shown in Figure 39.20.

Step 3.2. Choose *Others* in the Categories section and *Cascading Style Sheet* in the File Types section to display the New Cascading Style Sheet dialog box, as shown in Figure 39.21.

Step 3.3. Enter `tablestyle` as the File Name and click *Finish* to create tablestyle.css under the resources node.

Step 3.4. Define the CSS style as shown in Listing 39.15.



New Cascading Style Sheet

Steps

1. Choose File Type
2. **Name and Location**

Name and Location

File Name: tablestyle

Project: jsf2demo

Folder: web\resources

Created File: C:\book\jsf2demo\web\resources\tablestyle.css

FIGURE 39.21 The New Cascading Style Sheet dialog box creates a new style sheet file.

LISTING 39.13 CourseNameJSFBean.java

```java
package jsf2demo;

import java.sql.*;
import java.util.ArrayList;
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Named;

@Named(value = "courseName")
@ApplicationScoped
public class CourseNameJSFBean {
  private PreparedStatement studentStatement = null;
  private String choice; // Selected course
  private String[] titles; // Course titles

  /** Creates a new instance of CourseName */
  public CourseNameJSFBean() {
    initializeJdbc();
  }

  /** Initialize database connection */
  private void initializeJdbc() {
    try {
      Class.forName("com.mysql.jdbc.Driver");
      System.out.println("Driver loaded");

      // Connect to the sample database
      Connection connection = DriverManager.getConnection(
        "jdbc:mysql://localhost/javabook", "scott", "tiger");

      // Get course titles
      PreparedStatement statement = connection.prepareStatement(
        "select title from course");

      ResultSet resultSet = statement.executeQuery();

      // Store resultSet into array titles
      ArrayList<String> list = new ArrayList<>();
```

application scope

initialize JDBC

connect to database

get course titles

execute SQL

```
38        while (resultSet.next()) {
39          list.add(resultSet.getString(1));
40        }
41        titles = new String[list.size()]; // Array for titles        titles array
42        list.toArray(titles); // Copy strings from list to array
43
44        // Define a SQL statement for getting students
45        studentStatement = connection.prepareStatement(
46          "select Student.ssn, "
47          + "student.firstName, Student.mi, Student.lastName, "
48          + "Student.phone, Student.birthDate, Student.street, "
49          + "Student.zipCode, Student.deptId "
50          + "from Student, Enrollment, Course "
51          + "where Course.title = ? "
52          + "and Student.ssn = Enrollment.ssn "
53          + "and Enrollment.courseId = Course.courseId;");
54      }
55      catch (Exception ex) {
56        ex.printStackTrace();
57      }
58    }
59
60    public String[] getTitles() {
61      return titles;
62    }
63
64    public String getChoice() {
65      return choice;
66    }
67
68    public void setChoice(String choice) {
69      this.choice = choice;
70    }
71
72    public ResultSet getStudents() throws SQLException {        get students
73      if (choice == null) {
74        if (titles.length == 0)
75          return null;
76        else
77          studentStatement.setString(1, titles[0]);        set a default course
78      }
79      else {
80        studentStatement.setString(1, choice); // Set course title        set a course
81      }
82
83      // Get students for the specified course
84      return studentStatement.executeQuery();        return students
85    }
86  }
```

We use the same MySQL database **javabook** created in Chapter 34, "Java Database Programming." The scope for this managed bean is application. The bean is created when the project is launched from the server. The `initializeJdbc` method loads the JDBC driver for MySQL (lines 23 and 24), connects to the MySQL database (lines 27 and 28), creates statement for obtaining course titles (lines 31 and 32), and creates a statement for obtaining the student information for the specified course (lines 45–53). Lines 31–42 execute the statement for obtaining course titles and store them in array `titles`.

The `getStudents()` method returns a `ResultSet` that consists of all students enrolled in the specified course (lines 72–85). The choice for the title is set in the statement to obtain the student for the specified title (line 80). If choice is `null`, the first title in the titles array is set in the statement (line 77). If no titles in the course, `getStudents()` returns `null` (line 75).

add MySQL in the Libraries node

TIP
In order to use the MySQL database from this project, you have to add the MySQL JDBC driver from the Libraries node in the Project pane in NetBeans.

Listing 39.14 DisplayStudent.xhtml

style sheet

bind choice

titles

display button

bind result set

rowClasses

headerClass

styleClass

ssn column

firstName column

mi column

lastName column

phone column

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
  <h:head>
    <title>Display Student</title>
    <h:outputStylesheet name="tablestyle.css"/>
  </h:head>
  <h:body>
    <h:form>
      <h:outputLabel value="Choose a Course: " />
      <h:selectOneMenu value="#{courseName.choice}">
        <f:selectItems value="#{courseName.titles}" />
      </h:selectOneMenu>

      <h:commandButton style="margin-left: 20px"
                       value="Display Students" />

      <br /> <br />
      <h:dataTable value="#{courseName.students}" var="student"
                   rowClasses="oddTableRow, evenTableRow"
                   headerClass="tableHeader"
                   styleClass="table">
        <h:column>
          <f:facet name="header">SSN</f:facet>
          #{student.ssn}
        </h:column>

        <h:column>
          <f:facet name="header">First Name</f:facet>
          #{student.firstName}
        </h:column>

        <h:column>
          <f:facet name="header">MI</f:facet>
          #{student.mi}
        </h:column>

        <h:column>
          <f:facet name="header">Last Name</f:facet>
          #{student.lastName}
        </h:column>

        <h:column>
          <f:facet name="header">Phone</f:facet>
          #{student.phone}
        </h:column>

        <h:column>
```

```
52            <f:facet name="header">Birth Date</f:facet>
53            #{student.birthDate}
54          </h:column>
55
56          <h:column>
57            <f:facet name="header">Dept</f:facet>
58            #{student.deptId}
59          </h:column>
60        </h:dataTable>
61      </h:form>
62    </h:body>
63  </html>
```

birthDate column

deptId column

Line 9 specifies that the style sheet **tablestyle.css** created in Step 3 is used in this XMTHL file. The rowClasses = "oddTableRow, evenTableRow" attribute specifies the style applied to the rows alternately using oddTableRow and evenTableRow (line 23). The headerClasses = "tableHeader" attribute specifies that the tableHeader class is used for header style (line 24). The styleClasses = "table" attribute specifies that the table class is used for the style of all other elements in the table (line 25).

Line 14 binds the choice property in the courseName bean with the combo box. The selection values in the combo box are bound with the titles array property (line 15).

Line 22 binds the table value with a database result set using the attribute value = "#{courseName.students}". The var="student" attribute associates a row in the result set with student. Lines 26–59 specify the column values using student.ssn (line 28), student.firstName (line 33), student.mi (line 38), student.lastName (line 33), student.phone (line 48), student.birthDate (line 53), and student.deptId (line 58).

LISTING 39.15 tablestyle.css

```
/* Style for table */
.tableHeader {
  font-family:"Trebuchet MS", Arial, Helvetica, sans-serif;
  border-collapse:collapse;
  font-size:1.1em;
  text-align:left;
  padding-top:5px;
  padding-bottom:4px;
  background-color:#A7C942;
  color:white;
  border:1px solid #98bf21;
}

.oddTableRow {
  border:1px solid #98bf21;
}

.evenTableRow {
  background-color: #eeeeee;
  font-size:1em;

  padding:3px 7px 2px 7px;

  color:#000000;
  background-color:#EAF2D3;
}

.table {
  border:1px solid green;
}
```

tableHeader

oddTableRow

evenTableRow

table

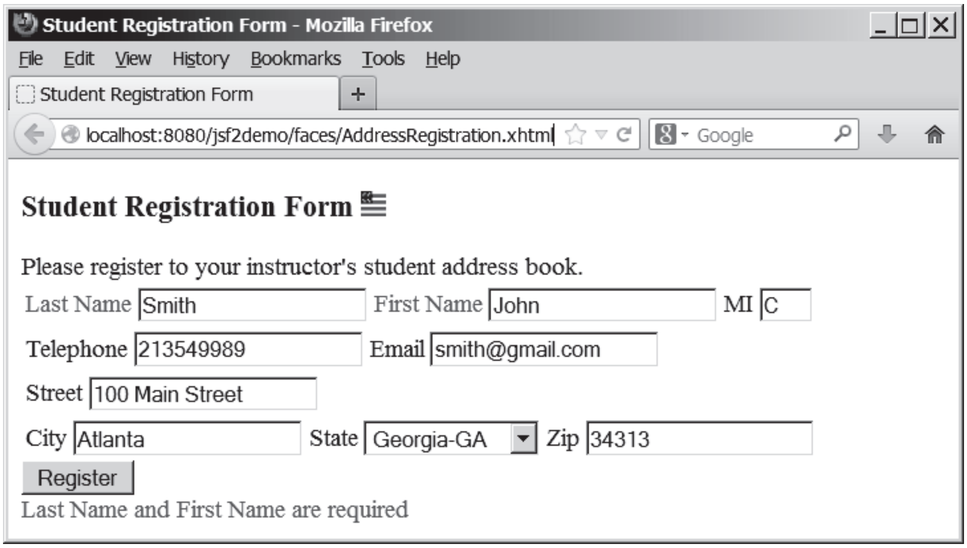
The style sheet file defines the style classes `tableHeader` (line 2) for table header style, `oddTableRow` for odd table rows (line 14), `evenTableRow` for even table rows (line 18), and table for all other table elements (line 28).

39.9 Opening New JSF Pages

Key Point

You can open new JSF pages from the current JSF pages.

All the examples you have seen so far use only one JSF page in a project. Suppose you want to register student information to the database. The application first displays the page as shown in Figure 39.22 to collect student information. After the user enters the information and clicks the *Submit* button, a new page is displayed to ask the user to confirm the input, as shown in Figure 39.23. If the user clicks the *Confirm* button, the data are stored into the database and the status page is displayed, as shown in Figure 39.24. If the user clicks the *Go Back* button, it goes back to the first page.


Student Registration Form - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Student Registration Form
localhost:8080/jsf2demo/faces/AddressRegistration.xhtml
Google

Student Registration Form

Please register to your instructor's student address book.

Last Name Smith First Name John MI C

Telephone 213549989 Email smith@gmail.com

Street 100 Main Street

City Atlanta State Georgia-GA Zip 34313

Register

Last Name and First Name are required

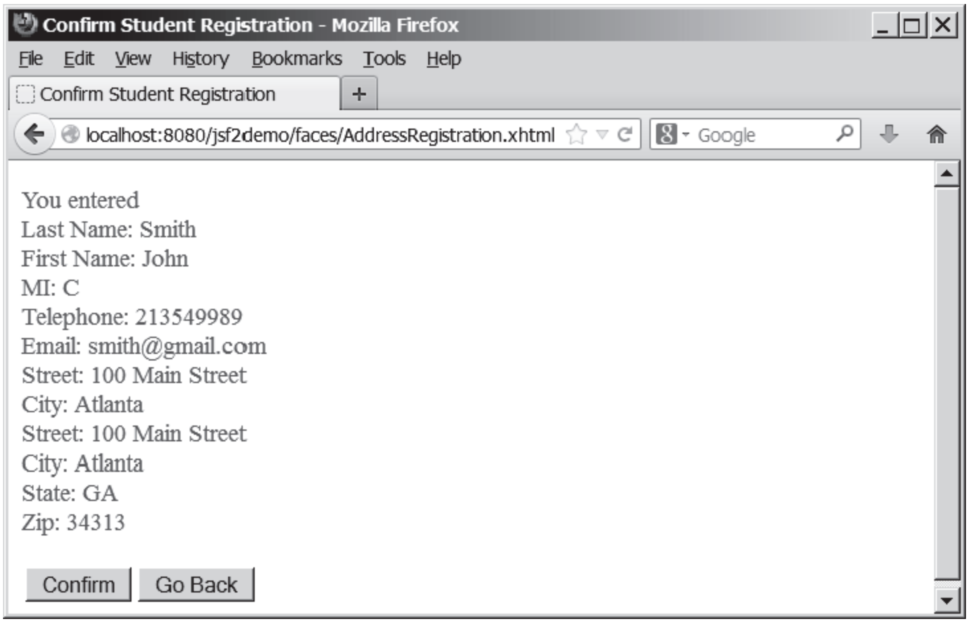
FIGURE 39.22 This page lets the user enter input.

For this project, you need to create three JSF pages named AddressRegistration.xhtml, ConfirmAddress.xhtml, and AddressStoredStatus.xhtml in Listings 39.16–39.18. The project starts with AddressRegistration.xhtml. When clicking the *Submit* button, the action for the button returns “ConfirmAddress” if the last name and first name are not empty, which causes ConfirmAddress.xhtml to be displayed. When clicking the *Confirm* button, the status page AddressStoredStatus.xhtml is displayed. When clicking the *Go Back* button, the first page AddressRegistration.xhtml is now displayed.

LISTING 39.16 AddressRegistration.xhtml

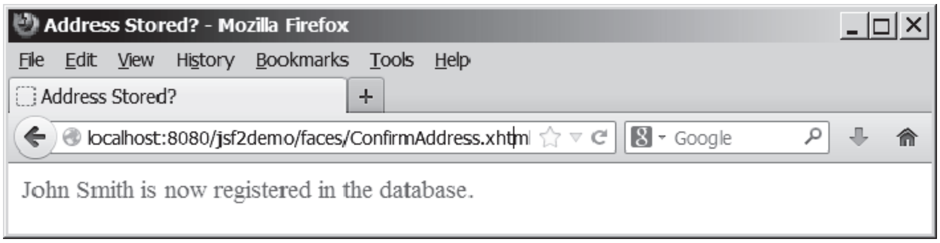
```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
  <h:head>
    <title>Student Registration Form</title>
```

jsf core namespace


Confirm Student Registration - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Confirm Student Registration
localhost:8080/jsf2demo/faces/AddressRegistration.xhtml
Google

You entered
Last Name: Smith
First Name: John
MI: C
Telephone: 213549989
Email: smith@gmail.com
Street: 100 Main Street
City: Atlanta
Street: 100 Main Street
City: Atlanta
State: GA
Zip: 34313

FIGURE 39.23 This page lets the user confirm the input.


Address Stored? - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Address Stored?
localhost:8080/jsf2demo/faces/ConfirmAddress.xhtml
Google

John Smith is now registered in the database.

FIGURE 39.24 This page displays the status of the user input.

```
</h:head>
<h:body>
  <h:form>
    <!-- Use h:graphicImage -->
    <h3>Student Registration Form
      <h:graphicImage name="usIcon.gif" library="image"/>
    </h3>

    Please register to your instructor's student address book.
    <!-- Use h:panelGrid -->
    <h:panelGrid columns="6">
      <h:outputLabel value="Last Name" style="color:red"/>
      <h:inputText id="lastNameInputText"
                   value="#{addressRegistration.lastName}"/>
      <h:outputLabel value="First Name" style="color:red"/>
      <h:inputText id="firstNameInputText"
                   value="#{addressRegistration.firstName}"/>
      <h:outputLabel value="MI" />
      <h:inputText id="miInputText" size="1"
                   value="#{addressRegistration.mi}"/>
    </h:panelGrid>

```

bind lastName

bind firstName

bind mi

bind telephone

bind email

bind street

bind city

bind state

bind zip

process register

```
      <h:panelGrid columns="4">
        <h:outputLabel value="Telephone"/>
        <h:inputText id="telephoneInputText"
                     value="#{addressRegistration.telephone}"/>
        <h:outputLabel value="Email"/>
        <h:inputText id="emailInputText"
                     value="#{addressRegistration.email}"/>
      </h:panelGrid>

      <h:panelGrid columns="4">
        <h:outputLabel value="Street"/>
        <h:inputText id="streetInputText"
                     value="#{addressRegistration.street}"/>
      </h:panelGrid>

      <h:panelGrid columns="6">
        <h:outputLabel value="City"/>
        <h:inputText id="cityInputText"
                     value="#{addressRegistration.city}"/>
        <h:outputLabel value="State"/>
        <h:selectOneMenu id="stateSelectOneMenu"
                         value="#{addressRegistration.state}">
          <f:selectItem itemLabel="Georgia-GA" itemValue="GA" />
          <f:selectItem itemLabel="Oklahoma-OK" itemValue="OK" />
          <f:selectItem itemLabel="Indiana-IN" itemValue="IN"/>
        </h:selectOneMenu>
        <h:outputLabel value="Zip"/>
        <h:inputText id="zipInputText"
                     value="#{addressRegistration.zip}"/>
      </h:panelGrid>

      <!-- Use command button -->
      <h:commandButton value="Register"
                       action="#{addressRegistration.processSubmit()}"/>
      <br />
      <h:outputText escape="false" style="color:red"
                    value="#{addressRegistration.requiredFields}" />
    </h:form>
  </h:body>
</html>
```

LISTING 39.17 ConfirmAddress.xhtml

process confirm

go to AddressRegistration page

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
  <h:head>
    <title>Confirm Student Registration</title>
  </h:head>
  <h:body>
    <h:form>
      <h:outputText escape="false" style="color:red"
                    value="#{registration1.input}" />
      <h:panelGrid columns="2">
      <h:commandButton value="Confirm"
         action = "#{registration1.storeStudent()}"/>
      <h:commandButton value="Go Back"
         action = "AddressRegistration"/>
      </h:panelGrid>
    </h:form>
  </h:body>
</html>
```

Listing 39.18 AddressStoredStatus.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
  <h:head>
    <title>Address Stored?</title>
  </h:head>
  <h:body>
    <h:form>
      <h:outputText escape="false" style="color:green"
                    value="#{registration1.status}" />
    </h:form>
  </h:body>
</html>
```

display status

Listing 39.19 AddressRegistrationJSFBean.java

```
package jsf2demo;

import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
import java.sql.*;
import java.io.Serializable;

@Named(value = "addressRegistration")
@SessionScoped
public class AddressRegistrationJSFBean implements Serializable {
  private String lastName;
  private String firstName;
  private String mi;
  private String telephone;
  private String email;
  private String street;
  private String city;
  private String state;
  private String zip;
  private String status = "Nothing stored";
  // Use a prepared statement to store a student into the database
  private PreparedStatement pstmt;

  public AddressRegistrationJSFBean() {
    initializeJdbc();
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

```

managed bean

session scope

property lastName

initialize database

```
  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getMi() {
    return mi;
  }

  public void setMi(String mi) {
    this.mi = mi;
  }

  public String getTelephone() {
    return telephone;
  }

  public void setTelephone(String telephone) {
    this.telephone = telephone;
  }

  public String getEmail() {
    return email;
  }

  public void setEmail(String email) {
    this.email = email;
  }

  public String getStreet() {
    return street;
  }

  public void setStreet(String street) {
    this.street = street;
  }

  public String getCity() {
    return city;
  }

  public void setCity(String city) {
    this.city = city;
  }

  public String getState() {
    return state;
  }

  public void setState(String state) {
    this.state = state;
  }

  public String getZip() {
    return zip;
  }

  public void setZip(String zip) {
    this.zip = zip;
  }

  private boolean isRquiredFieldsFilled() {
    return !(lastName == null || firstName == null
           || lastName.trim().length() == 0
           || firstName.trim().length() == 0);
  }

  public String processSubmit() {
    if (isRquiredFieldsFilled())
      return "ConfirmAddress";                              go to a new page
    else
      return "";
  }

  public String getRequiredFields() {
    if (isRquiredFieldsFilled())                            check required fields
      return "";
    else
      return "Last Name and First Name are required";
  }

  public String getInput() {                                get input
    return "<p style=\"color:red\">You entered <br />"
            + "Last Name: " + lastName + "<br />"
            + "First Name: " + firstName + "<br />"
            + "MI: " + mi + "<br />"
            + "Telephone: " + telephone + "<br />"
            + "Email: " + email + "<br />"
            + "Street: " + street + "<br />"
            + "City: " + city + "<br />"
            + "Street: " + street + "<br />"
            + "City: " + city + "<br />"
            + "State: " + state + "<br />"
            + "Zip: " + zip + "</p>";
  }

  /** Initialize database connection */
  private void initializeJdbc() {
    try {
      // Explicitly load a MySQL driver
      Class.forName("com.mysql.jdbc.Driver");
      System.out.println("Driver loaded");

      // Establish a connection
      Connection conn = DriverManager.getConnection(
             "jdbc:mysql://localhost/javabook", "scott", "tiger");

      // Create a Statement
      pstmt = conn.prepareStatement("insert into Address (lastName,"
             + " firstName, mi, telephone, email, street, city, "
             + "state, zip) values (?, ?, ?, ?, ?, ?, ?, ?, ?)");
    }
    catch (Exception ex) {
      System.out.println(ex);
    }
  }

  /** Store an address to the database */
  public String storeStudent() {                            store address
```

update status

go to a new page

```
158       try {
159         pstmt.setString(1, lastName);
160         pstmt.setString(2, firstName);
161         pstmt.setString(3, mi);
162         pstmt.setString(4, telephone);
163         pstmt.setString(5, email);
164         pstmt.setString(6, street);
165         pstmt.setString(7, city);
166         pstmt.setString(8, state);
167         pstmt.setString(9, zip);
168         pstmt.executeUpdate();
169         status = firstName + " " + lastName
170                 + " is now registered in the database.";
171       }
172       catch (Exception ex) {
173         status = ex.getMessage();
174       }
175
176       return "AddressStoredStatus";
177     }
178
179     public String getStatus() {
180       return status;
181     }
182   }
```

A session-scoped managed bean must implement the `java.io.Serializable` interface. So, the `AddressRegistration` class is defined as a subtype of `java.io.Serializable`.

The action for the *Register* button in the `AddressRegistration` JSF page is `processSubmit()` (line 64 in AddressRegistration.xhtml). This method checks if last name and first name are not empty (lines 106–111 in AddressRegistrationJSFBean.java). If so, it returns a string `"ConfirmAddress"`, which causes the `ConfirmAddress` JSF page to be displayed.

The `ConfirmAddress` JSF page displays the data entered from the user (line 12 in ConfirmAddress.xhtml). The `getInput()` method (lines 120–133 in AddressRegistrationJSFBean.java) collects the input.

The action for the *Confirm* button in the `ConfirmAddress` JSF page is `storeStudent()` (line 15 in ConfirmAddress.xhtml). This method stores the address in the database (lines 157–177 in AddressRegistrationJSFBean.java) and returns a string `"AddressStoredStatus"`, which causes the `AddressStoredStatus` page to be displayed. The status message is displayed in this page (line 12 in AddressStoredStatus.xhtml).

The action for the *Go Back* button in the `ConfirmAddress` page is `"AddressRegistration"` (line 17 in ConfirmAddress.xhtml). This causes the `AddressRegistration` page to be displayed for the user to reenter the input.

The scope of the managed bean is session (line 9 AddressRegistrationJSFBean.java) so the multiple pages can share the same bean.

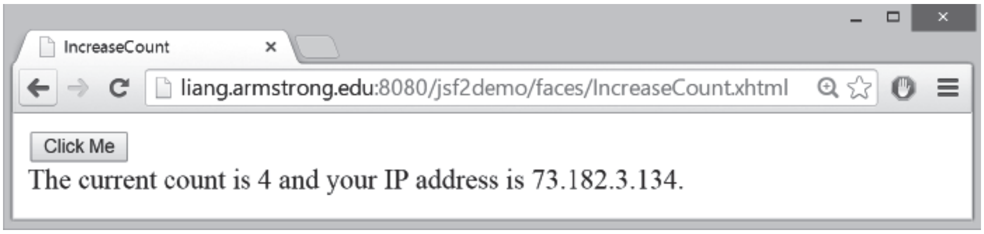
Note this program loads the database driver explicitly (line 139 AddressRegistrationJSFBean.java). Sometimes, an IDE such as NetBeans is not able to find a suitable driver. Loading a driver explicitly can avoid this problem.

39.10 Contexts and Dependency Injection

Key Point

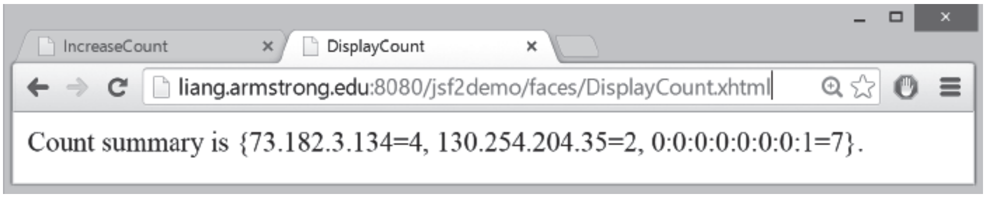
Contexts and dependency injection enables beans to be shared in multiple applications.

Contexts and dependency injection, short for *CDI*, allows multiple programs to share a bean. To illustrate the need for this, consider two simple webpages and a server object named `track`. One page contains a button and a message that displays the number of times the button is clicked from the current IP address, as shown in Figure 39.25. When the button is clicked for the first time, the user's IP address along with count value 1 is stored in a map with the IP address as the key. When the button is clicked again, the count value for the IP address is increased in the map. The other page simply displays the total count from each IP address, as shown in Figure 39.26. The Track class is defined as shown in Listing 39.20.



The current count is 4 and your IP address is 73.182.3.134.

FIGURE 39.25 The count is updated when the *Click Me* button is clicked.



Count summary is {73.182.3.134=4, 130.254.204.35=2, 0:0:0:0:0:0:0:1=7}.

FIGURE 39.26 The count for each client IP Address is displayed.

LISTING 39.20 Track.java

```
 1  package jsf2demo;
 2
 3  import java.util.HashMap;
 4  import java.util.Map;
 5  import javax.enterprise.context.ApplicationScoped;
 6
 7  @ApplicationScoped
 8  public class Track {
 9    private Map<String, Integer> map = new HashMap<>();
10
11    public void add(String ipAddress) {
12      map.put(ipAddress, map.containsKey(ipAddress) ?
13        map.get(ipAddress) + 1 : 1);
14    }
15
16    public int getCount(String ipAddress) {
17      return map.containsKey(ipAddress) ? map.get(ipAddress) : 0;
18    }
19
20    public String getAllCount() {
21      return "Count summary is " + map;
22    }
23  }
```

application scope

store counts

add or update count

return count

return all counts

A Track object uses a map to store an IP address and its count with IP address as a key (line 9). The add method (lines 11–14) adds an IP Address to the map. If the IP address is not in the map, a new entry is created for the IP Address with value 1. Otherwise, the value for the IP address is incremented by 1 in the map. The getCount method (lines 16–18) returns the count for an IP address. If the IP address is not in the map, the method returns 0. The getAllCount method (lines 20–22) simply returns a string that describes the counts for all IP address in the map.

We now create a page named **IncreaseCount.xhtml** (Listing 39.21) with a button for displaying the number of times a button is clicked on the client, and create a page named **DisplayCount.xhtml** (Listing 39.22) for displaying the counts from all clients.

Listing 39.21 IncreaseCount.xhtml

process a click
obtain count
obtain IP address

```
 1  <?xml version='1.0' encoding='UTF-8' ?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
 4  <html xmlns="http://www.w3.org/1999/xhtml"
 5        xmlns:h="http://xmlns.jcp.org/jsf/html">
 6    <h:head>
 7      <title>IncreaseCount</title>
 8    </h:head>
 9    <h:body>
10      <h:form>
11        <h:commandButton
12          action="#{increaseCount.click()}" value="Click Me"/>
13        <br>The current count is #{increaseCount.getCount()} and your
14          IP address is #{increaseCount.getIpAddress()}.</br>
15      </h:form>
16    </h:body>
17  </html>
```

Listing 39.22 DisplayCount.xhtml

obtain all counts

```
 1  <?xml version='1.0' encoding='UTF-8' ?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
 4  <html xmlns="http://www.w3.org/1999/xhtml"
 5        xmlns:h="http://xmlns.jcp.org/jsf/html">
 6    <h:head>
 7      <title>DisplayCount</title>
 8    </h:head>
 9    <h:body>
10      #{displayCount.getAllCount()}.
11    </h:body>
12  </html>
```

The IncreaseCount page uses the increasCount bean to process the click action (line 12), obtain the click count (line 13), and the client's IP address (line 14). The DisplayCount page uses the displayCount bean to obtain the count from all clients (line 10). Both increasCount and displayCount need to access the same Track object. How can you create a Track object to be used by different objects? JSF supports context dependency injection (CDI) for injecting an object into a class using the @Inject annotation. Listing 39.23 gives the implementation for IncreaseCount.java and Listing 39.24 for DisplayCount.java.

Listing 39.23 IncreaseCount.java

```
package jsf2demo;

import javax.enterprise.context.SessionScoped;
import javax.inject.Named;
import javax.faces.context.FacesContext;
import javax.inject.Inject;
import javax.servlet.http.HttpServletRequest;

@Named(value = "increaseCount")
```

```
10  @SessionScoped
11  public class IncreaseCount implements java.io.Serializable {
12    @Inject private Track track;
13    private String ipAddress;
14
15    public IncreaseCount() {
16      HttpServletRequest request = (HttpServletRequest)FacesContext
17        .getCurrentInstance().getExternalContext().getRequest();
18      this.ipAddress = request.getRemoteAddr();
19    }
20
21    public void click() {
22      track.add(ipAddress);
23    }
24
25    public String getIpAddress() {
26      return ipAddress;
27    }
28
29    public int getCount() {
30      return track.getCount(ipAddress);
31    }
32  }
```

session scope

inject track

increase count

obtain client's IP

add an IP address

count for an IP

Listing 39.24 DisplayCount.java

```
 1  package jsf2demo;
 2
 3  import javax.enterprise.context.ApplicationScoped;
 4  import javax.inject.Named;
 5  import javax.inject.Inject;
 6
 7  @Named(value = "displayCount")
 8  @ApplicationScoped
 9  public class DisplayCount {
10    @Inject private Track track;
11
12    public String getAllCount() {
13      return track.getAllCount();
14    }
15  }
```

application scope

inject track

obtain all counts

The `@Inject` annotation in line 12 of **IncreaseCount.java** and line 10 of **DisplayCount.java** injects a `Track` object. This `Track` object is created by the Java server container. The `track` data fields in both classes refer to this object.

In **IncreaseCount.java**, the constructor obtains the IP address of a client (lines 16 and 17) and sets it in the data field `ipAddress` (line 18). The `click` method adds the `ipAddress` to the map in the track object (line 22).

Note the scope for `Track` and `DisplayCount` is `ApplicationScoped` since these two objects are created once for the entire application. However, the scope for `IncreaseCount` is `SessionScoped` since each session has its own IP Address.

Key Terms

application scope 39-21
contexts and dependency injection (CDI) 39-40
JavaBean 39-5
request scope 39-21
scope 39-21
session scope 39-21
view scope 39-21

Chapter Summary

1. JSF enables you to completely separate Java code from HTML.

2. A `facelet` is an XHTML page that mixes JSF tags with XHTML tags.

3. JSF applications are developed using the Model-View-Controller (MVC) architecture, which separates the application's data (contained in the model) from the graphical presentation (the view).

4. The controller is the JSF framework that is responsible for coordinating interactions between view and the model.

5. In JSF, the facelets are the view for presenting data. Data are obtained from Java objects. Objects are defined using Java classes.

6. In JSF, the objects that are accessed from a facelet are JavaBeans objects.

7. The JSF expression can either use the property name or invoke the method to obtain the current time.

8. JSF provides many elements for displaying GUI components. The tags with the `h` prefix are in the JSF HTML Tag library. The tags with the `f` prefix are in the JSF Core Tag library.

9. You can specify the JavaBeans objects at the application scope, session scope, view scope, or request scope.

10. The view scope keeps the bean alive as long as you stay on the view. The view scope is between session and request scopes.

11. JSF provides several convenient and powerful ways for input validation. You can use the standard validator tags in the JSF Core Tag Library or create custom validators.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

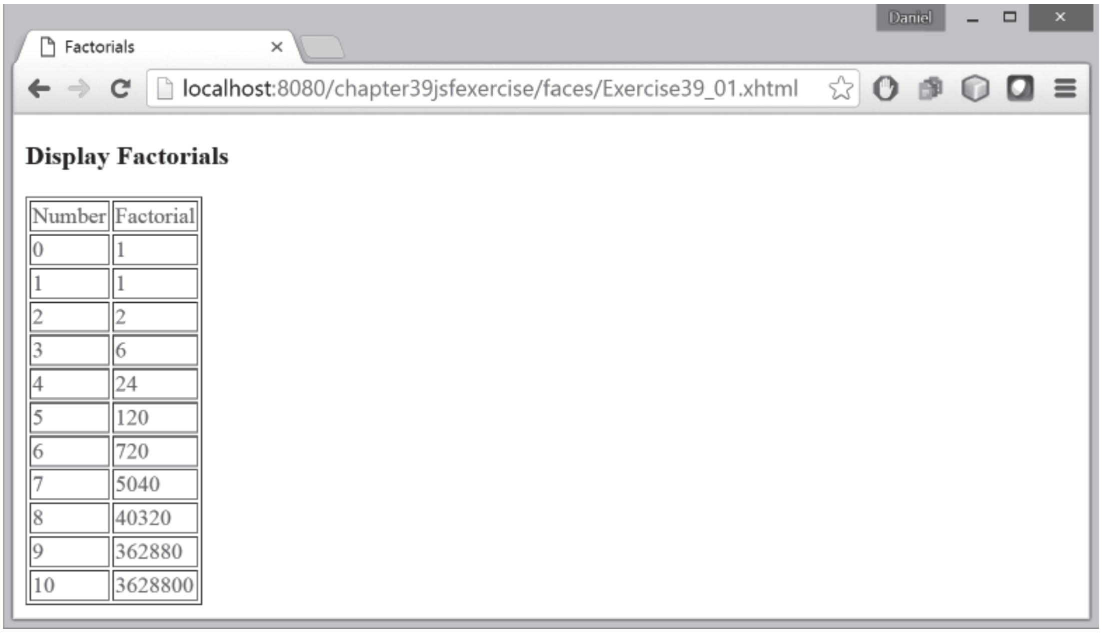
Programming Exercises

***39.1** *(Factorial table in JSF)* Write a JSF page that displays a factorial page as shown in Figure 39.27. Display the table in an `h:outputText` component. Set its `escape` property to `false` to display it as HTML contents.

***39.2** (*Multiplication table*) Write a JSF page that displays a multiplication table as shown in Figure 39.28.

***39.3** (*Calculate tax*) Write a JSF page to let the user enter taxable income and filing status, as shown in Figure 39.29a. Clicking the *Compute Tax* button computes and displays the tax, as shown in Figure 39.29b. Use the `computeTax` method introduced in Listing 3.5, ComputeTax.java, to compute tax.

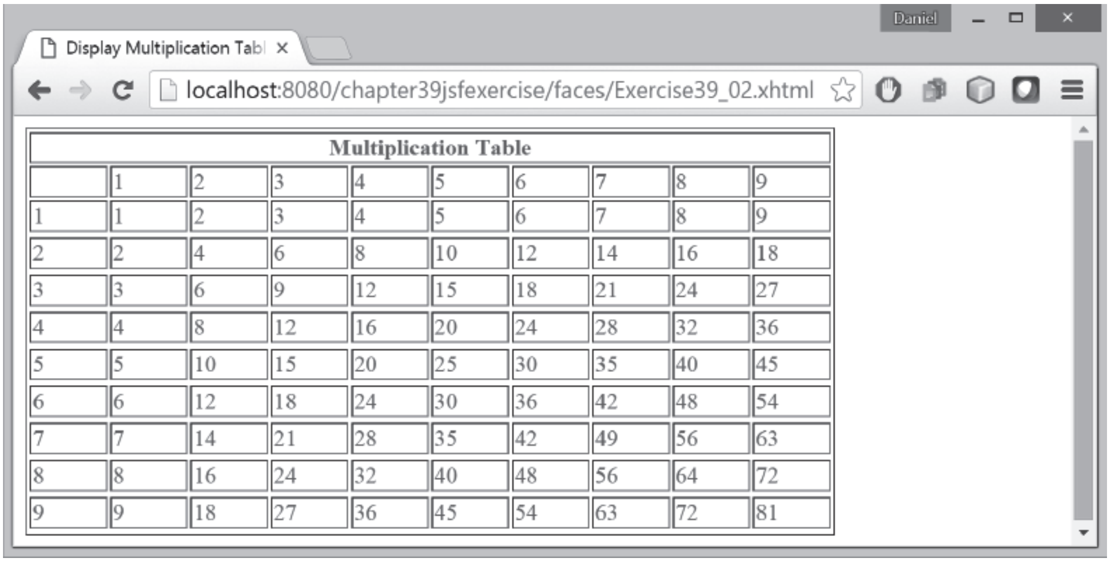
***39.4** (*Calculate loan*) Write a JSF page that lets the user enter loan amount, interest rate, and number of years, as shown in Figure 39.30a. Click the *Compute Loan Payment* button to compute and display the monthly and total loan payments, as shown in Figure 39.30b. Use the `Loan` class given in Listing 10.2, Loan.java, to compute the monthly and total payments.


Daniel
Factorials
localhost:8080/chapter39jsfexercise/faces/Exercise39_01.xhtml

Display Factorials

Number	Factorial
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800

Figure 39.27 The JSF page displays factorials for the numbers from 0 to 10 in a table.


Daniel
Display Multiplication Tabl
localhost:8080/chapter39jsfexercise/faces/Exercise39_02.xhtml

Multiplication Table

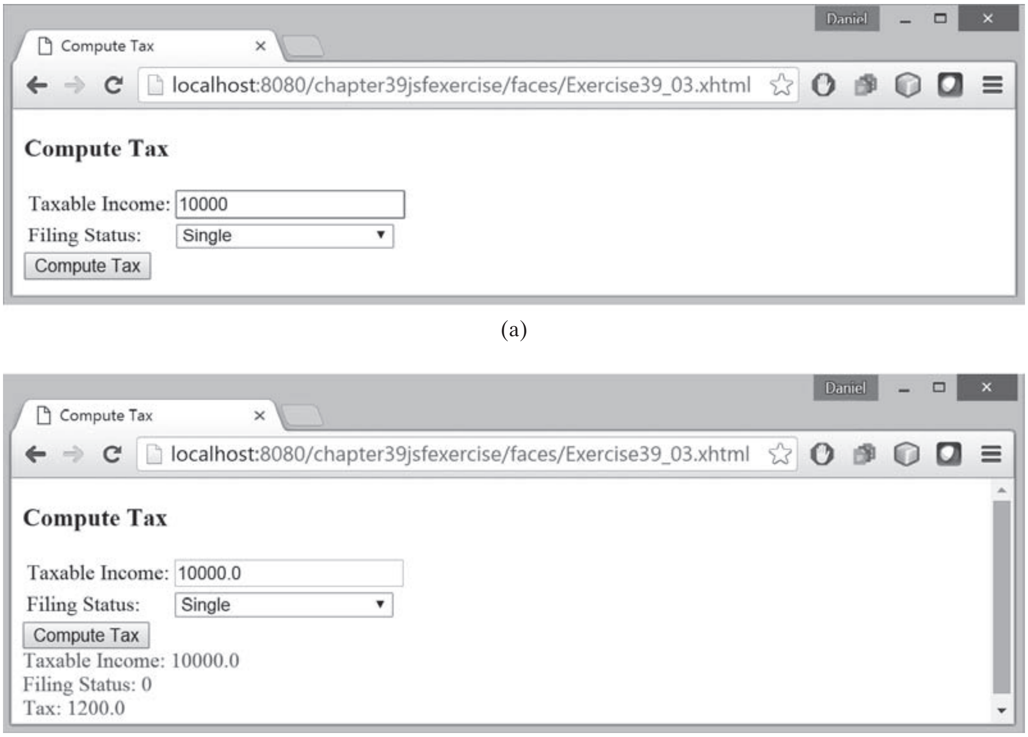
	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

Figure 39.28 The JSF page displays the multiplication table.

***39.5** (*Addition quiz*) Write a JSF program that generates addition quizzes randomly, as shown in Figure 39.31a. After the user answers all questions, it displays the result, as shown in Figure 39.31b.

***39.6** (*Large factorial*) Rewrite Exercise 39.1 to handle large factorial as shown in Figure 39.32. Use the `BigInteger` class introduced in Section 10.9.

***39.7** (*Guess birthday*) Listing 4.3, GuessBirthday.java, gives a program for guessing a birthday. Write a JSF program that displays five sets of numbers, as shown in Figure 39.33a. After the user checks the appropriate boxes and clicks the *Guess Birthday* button, the program displays the birthday, as shown in Figure 39.33b.


Compute Tax
localhost:8080/chapter39jsfexercise/faces/Exercise39_03.xhtml
Daniel

Compute Tax

Taxable Income: 10000
Filing Status: Single
Compute Tax

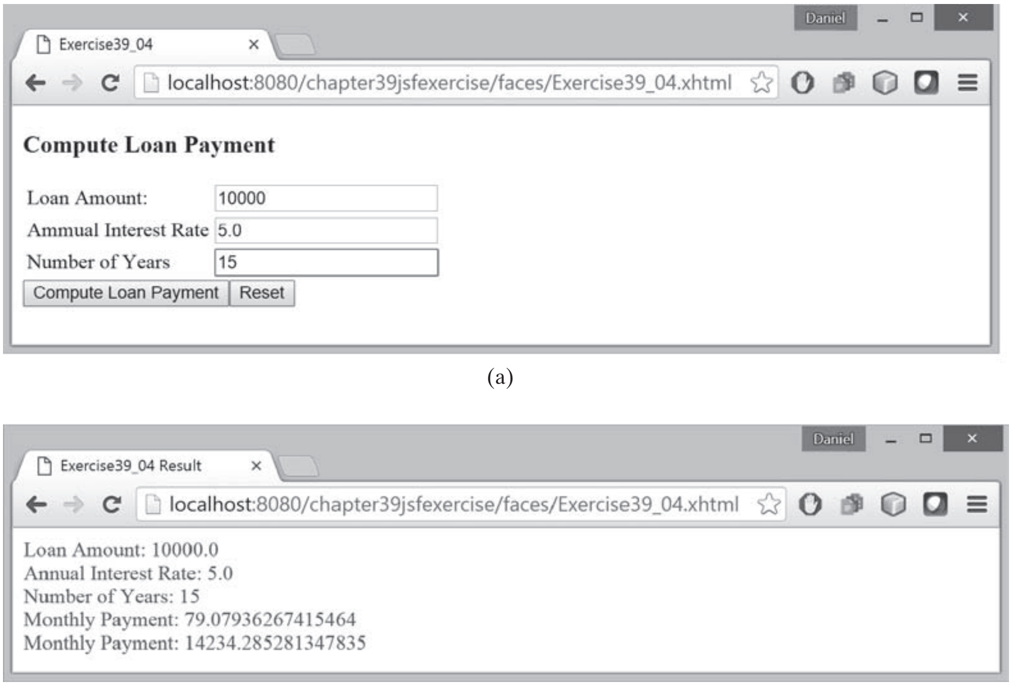
(a)

Compute Tax

Taxable Income: 10000.0
Filing Status: Single
Compute Tax
Taxable Income: 10000.0
Filing Status: 0
Tax: 1200.0

(b)

FIGURE 39.29 The JSF page computes the tax.


Exercise39_04
localhost:8080/chapter39jsfexercise/faces/Exercise39_04.xhtml
Daniel

Compute Loan Payment

Loan Amount: 10000
Ammual Interest Rate 5.0
Number of Years 15
Compute Loan Payment Reset

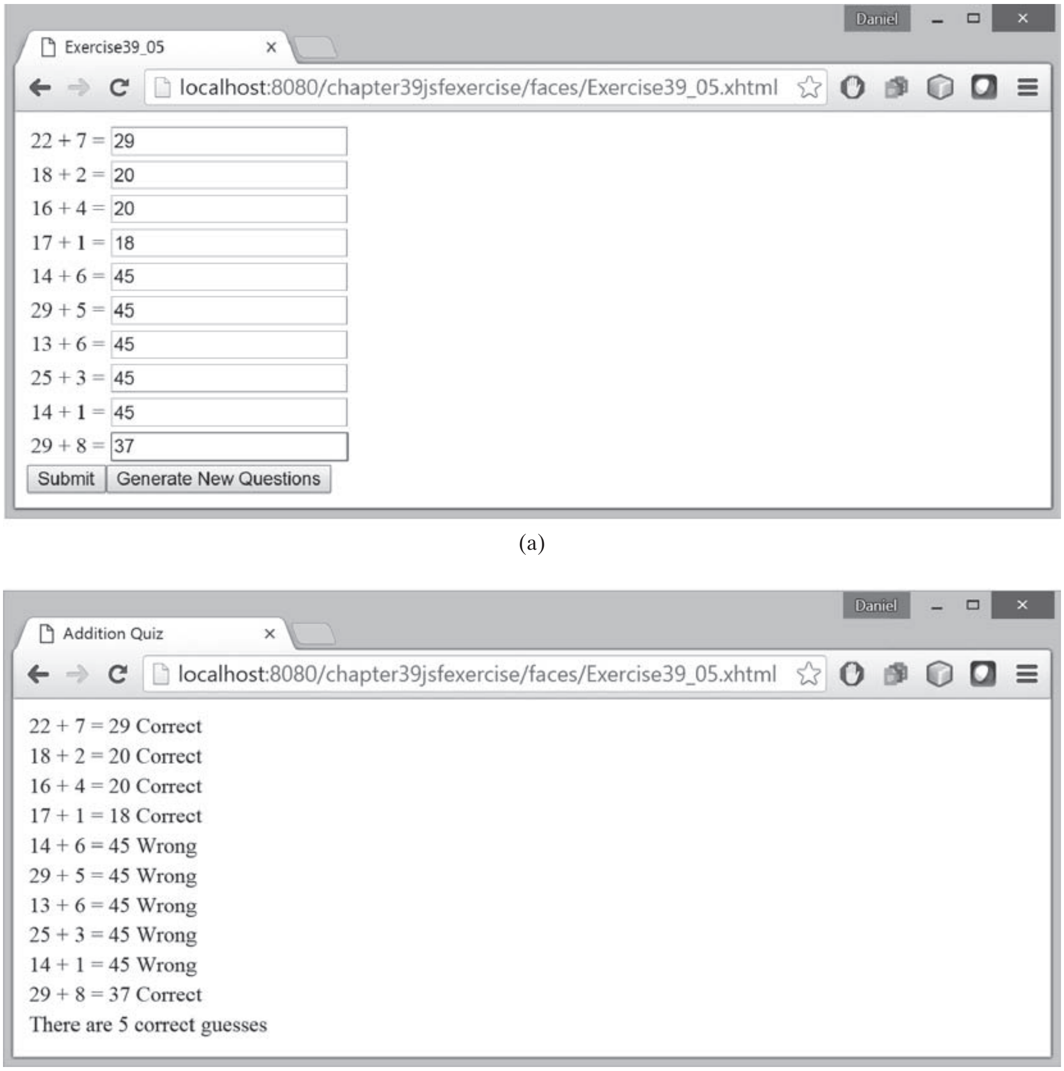
(a)

Exercise39_04 Result
localhost:8080/chapter39jsfexercise/faces/Exercise39_04.xhtml
Daniel

Loan Amount: 10000.0
Annual Interest Rate: 5.0
Number of Years: 15
Monthly Payment: 79.07936267415464
Monthly Payment: 14234.285281347835

(b)

FIGURE 39.30 The JSF page computes the loan payment.


Daniel
Exercise39_05
localhost:8080/chapter39jsfexercise/faces/Exercise39_05.xhtml
Submit
Generate New Questions
Addition Quiz

22 + 7 = 29
18 + 2 = 20
16 + 4 = 20
17 + 1 = 18
14 + 6 = 45
29 + 5 = 45
13 + 6 = 45
25 + 3 = 45
14 + 1 = 45
29 + 8 = 37

(a)

22 + 7 = 29 Correct
18 + 2 = 20 Correct
16 + 4 = 20 Correct
17 + 1 = 18 Correct
14 + 6 = 45 Wrong
29 + 5 = 45 Wrong
13 + 6 = 45 Wrong
25 + 3 = 45 Wrong
14 + 1 = 45 Wrong
29 + 8 = 37 Correct
There are 5 correct guesses

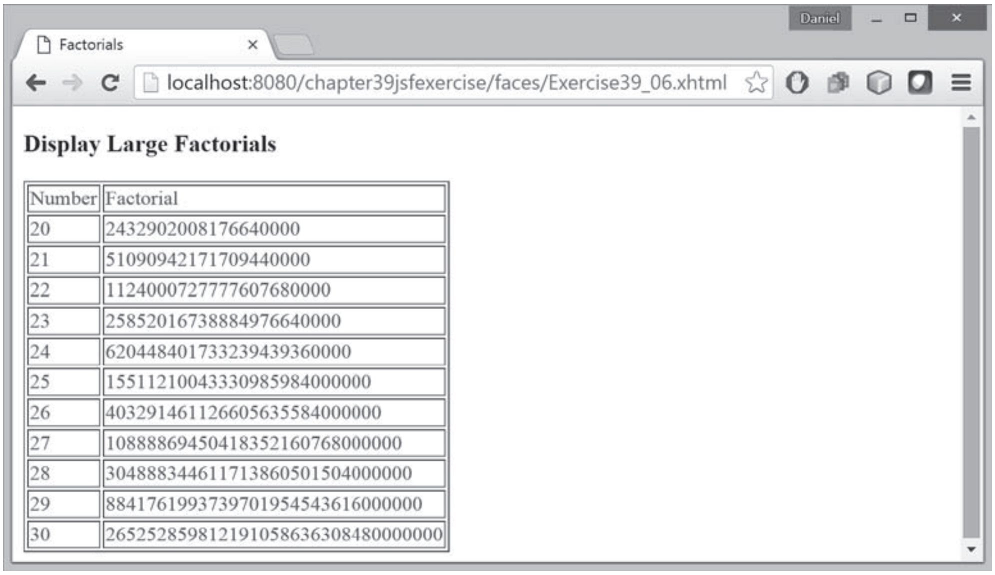
(b)

FIGURE 39.31 The program displays addition questions in (a) and answers in (b).

***39.8** (*Guess capitals*) Write a JSF that prompts the user to enter a capital for a state, as shown in Figure 39.34a. Upon receiving the user input, the program reports whether the answer is correct, as shown in Figure 39.34b. You can click the *Next* button to display another question. You can use a two-dimensional array to store the states and capitals, as proposed in Exercise 8.37. Create a list from the array and apply the `shuffle` method to reorder the list so the questions will appear in random order.

***39.9** (*Access and update a `Staff` table*) Write a JSF program that views, inserts, and updates staff information stored in a database, as shown in Figure 39.35. The view button displays a record with a specified ID. The `Staff` table is created as follows:

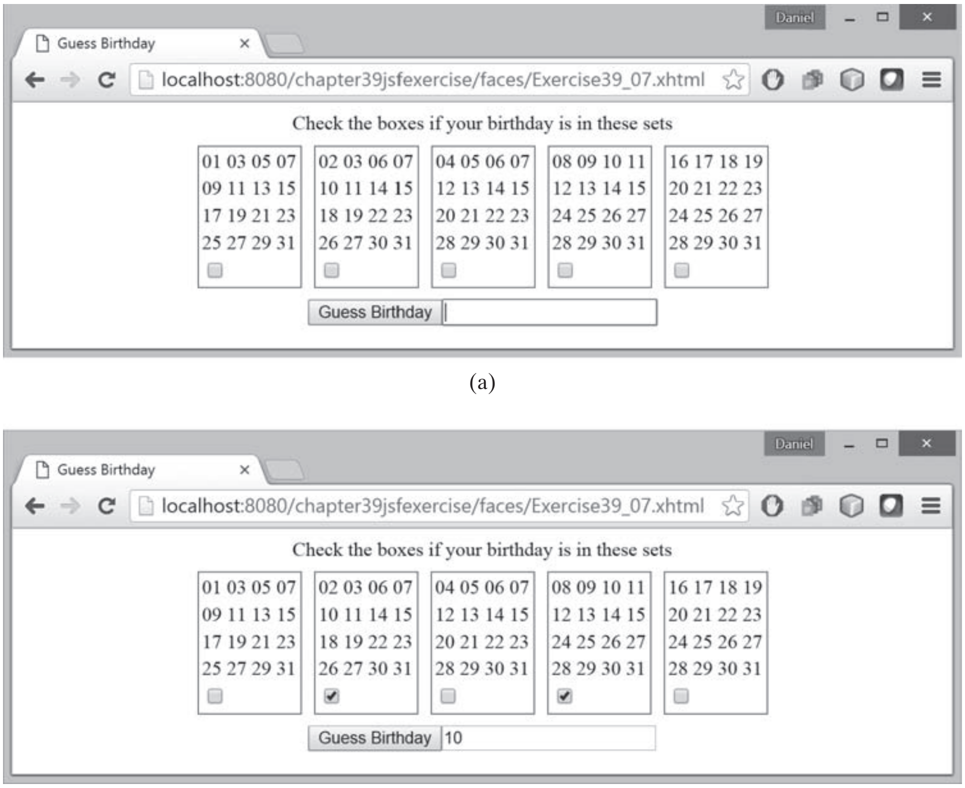
```
create table Staff (
  id char(9) not null,
  lastName varchar(15),
  firstName varchar(15),
```


Factorials
localhost:8080/chapter39jsfexercise/faces/Exercise39_06.xhtml
Daniel

Display Large Factorials

Number	Factorial
20	2432902008176640000
21	51090942171709440000
22	1124000727777607680000
23	25852016738884976640000
24	620448401733239439360000
25	15511210043330985984000000
26	403291461126605635584000000
27	10888869450418352160768000000
28	304888344611713860501504000000
29	8841761993739701954543616000000
30	265252859812191058636308480000000

Figure 39.32 The JSF page displays factorials for the numbers from 10 to 20 in a table.


Guess Birthday
localhost:8080/chapter39jsfexercise/faces/Exercise39_07.xhtml
Daniel

Check the boxes if your birthday is in these sets

01 03 05 07 09 11 13 15 17 19 21 23 25 27 29 31	02 03 06 07 10 11 14 15 18 19 22 23 26 27 30 31	04 05 06 07 12 13 14 15 20 21 22 23 28 29 30 31	08 09 10 11 12 13 14 15 24 25 26 27 28 29 30 31	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
□	□	□	□	□

Guess Birthday

(a)

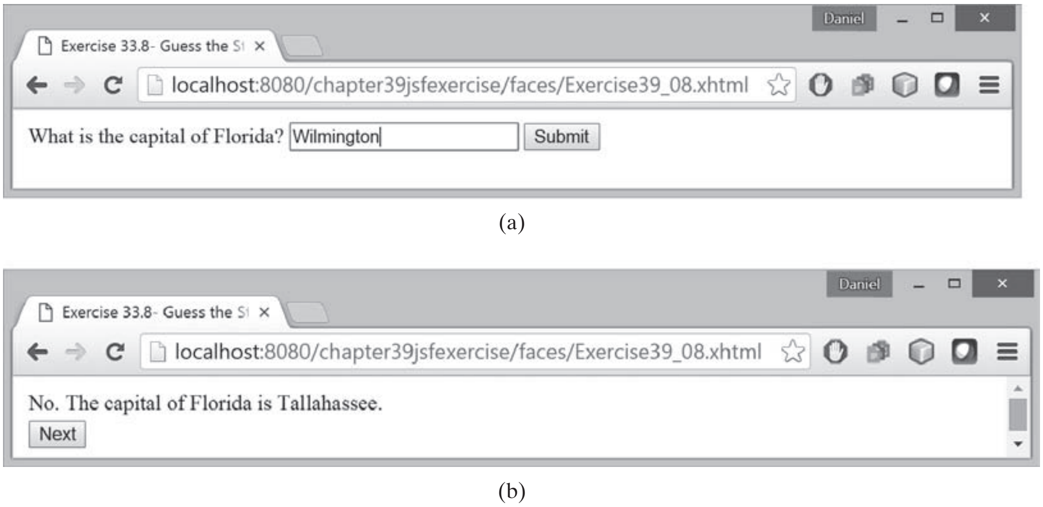
Check the boxes if your birthday is in these sets

01 03 05 07 09 11 13 15 17 19 21 23 25 27 29 31	02 03 06 07 10 11 14 15 18 19 22 23 26 27 30 31	04 05 06 07 12 13 14 15 20 21 22 23 28 29 30 31	08 09 10 11 12 13 14 15 24 25 26 27 28 29 30 31	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
□	☑	□	☑	□

Guess Birthday 10

(b)

Figure 39.33 (a) The program displays five sets of numbers for the user to check the boxes. (b) The program displays the date.


Exercise 33.8- Guess the S
localhost:8080/chapter39jsfexercise/faces/Exercise39_08.xhtml
Daniel

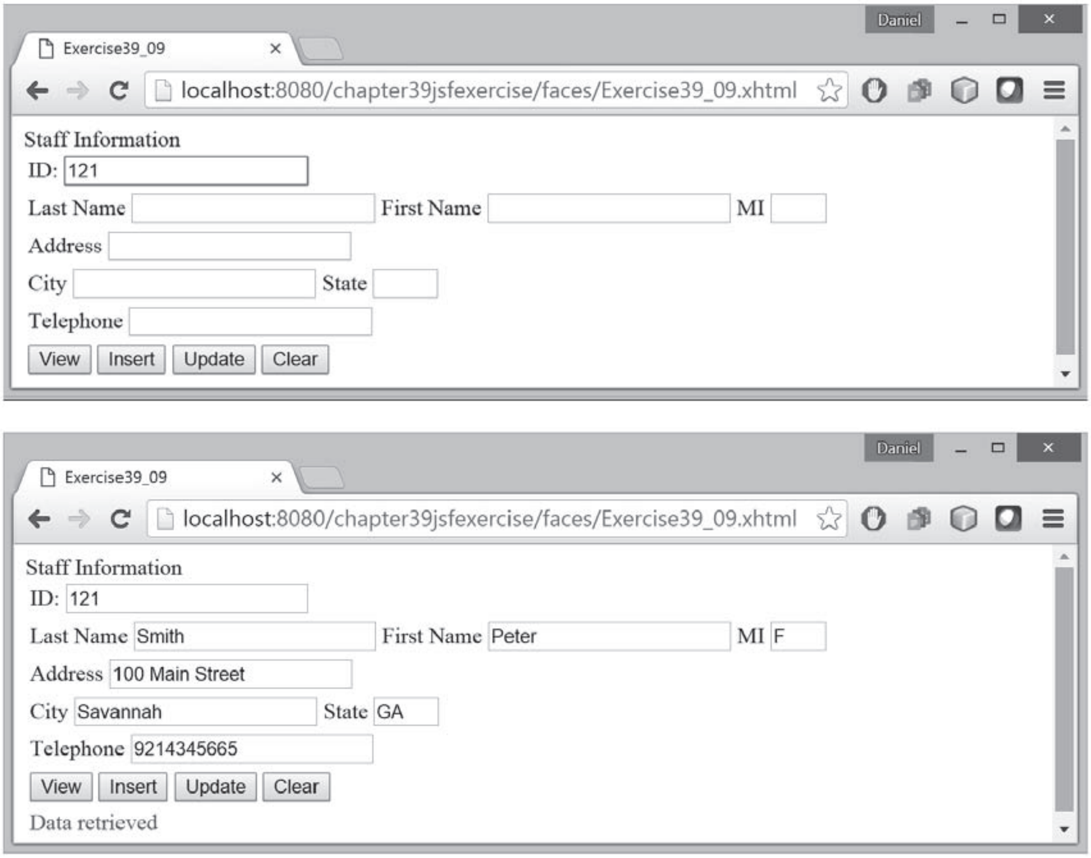
What is the capital of Florida? Wilmington

(a)

No. The capital of Florida is Tallahassee.

(b)

FIGURE 39.34 (a) The program displays a question. (b) The program displays the answer to the question.


Exercise39_09
localhost:8080/chapter39jsfexercise/faces/Exercise39_09.xhtml
Daniel

Staff Information
ID: 121
Last Name First Name MI
Address
City State
Telephone

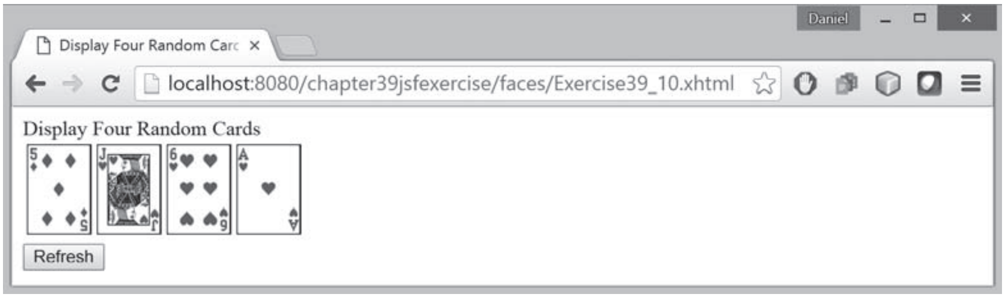
Staff Information
ID: 121
Last Name Smith First Name Peter MI F
Address 100 Main Street
City Savannah State GA
Telephone 9214345665

Data retrieved

FIGURE 39.35 The webpage lets you view, insert, and update staff information.

```
  mi char(1),
  address varchar(20),
  city varchar(20),
  state char(2),
  telephone char(10),
  email varchar(40),
  primary key (id)
);
```

***39.10** (*Random cards*) Write a JSF that displays four random cards from a deck of 52 cards, as shown in Figure 39.36. When the user clicks the *Refresh* button, four new random cards are displayed.

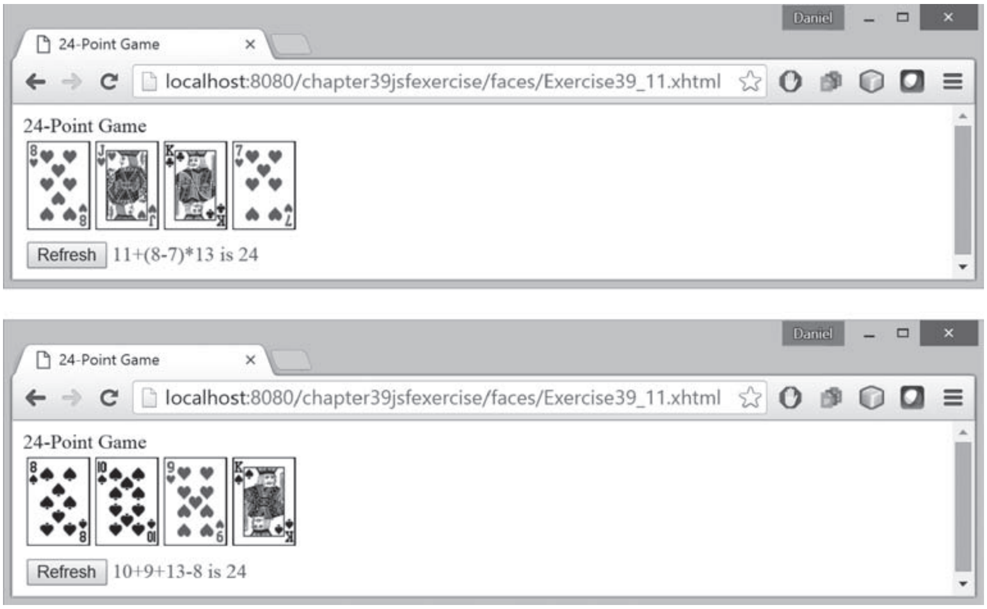

Daniel
Display Four Random Carc ×
localhost:8080/chapter39jsfexercise/faces/Exercise39_10.xhtml

Display Four Random Cards

Refresh

FIGURE 39.36 This JSF application displays four random cards.

*****39.11** (*Game: the 24-point card game*) Rewrite Exercise 20.13 using JSF, as shown in Figure 39.37. Upon clicking the *Refresh* button, the program displays four random cards and displays an expression if a 24-point solution exists. Otherwise, it displays “No solution”.


Daniel
24-Point Game ×
localhost:8080/chapter39jsfexercise/faces/Exercise39_11.xhtml

24-Point Game

Refresh 11+(8-7)*13 is 24

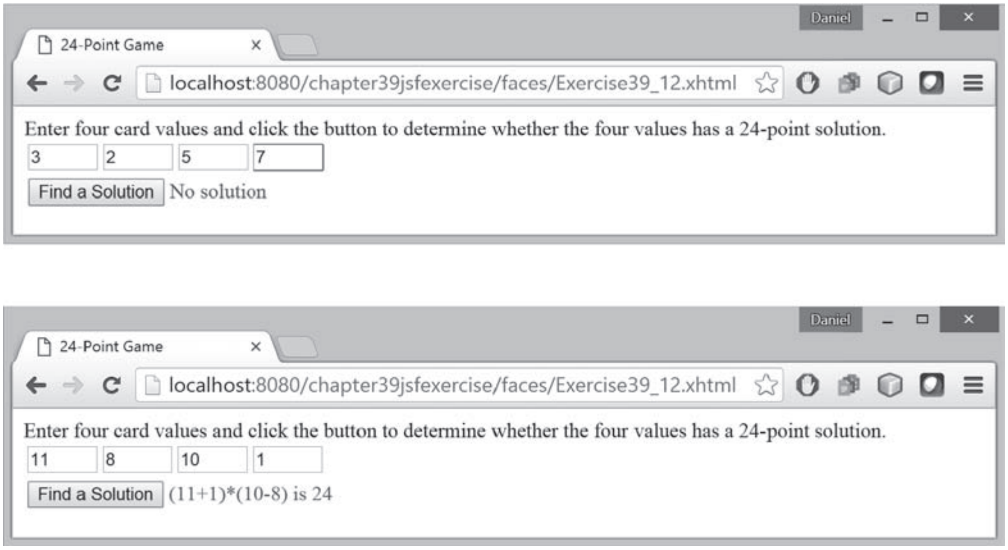
Daniel
24-Point Game ×
localhost:8080/chapter39jsfexercise/faces/Exercise39_11.xhtml

24-Point Game

Refresh 10+9+13-8 is 24

FIGURE 39.37 The JSF application solves a 24-point card game.

*****39.12** (*Game: the 24-point card game*) Rewrite Exercise 20.17 using JSF, as shown in Figure 39.38. The program lets the user enter four card values and finds a solution upon clicking the *Find a Solution* button.


24-Point Game

localhost:8080/chapter39jsfexercise/faces/Exercise39_12.xhtml

Enter four card values and click the button to determine whether the four values has a 24-point solution.

3 | 2 | 5 | 7

Find a Solution No solution

24-Point Game

localhost:8080/chapter39jsfexercise/faces/Exercise39_12.xhtml

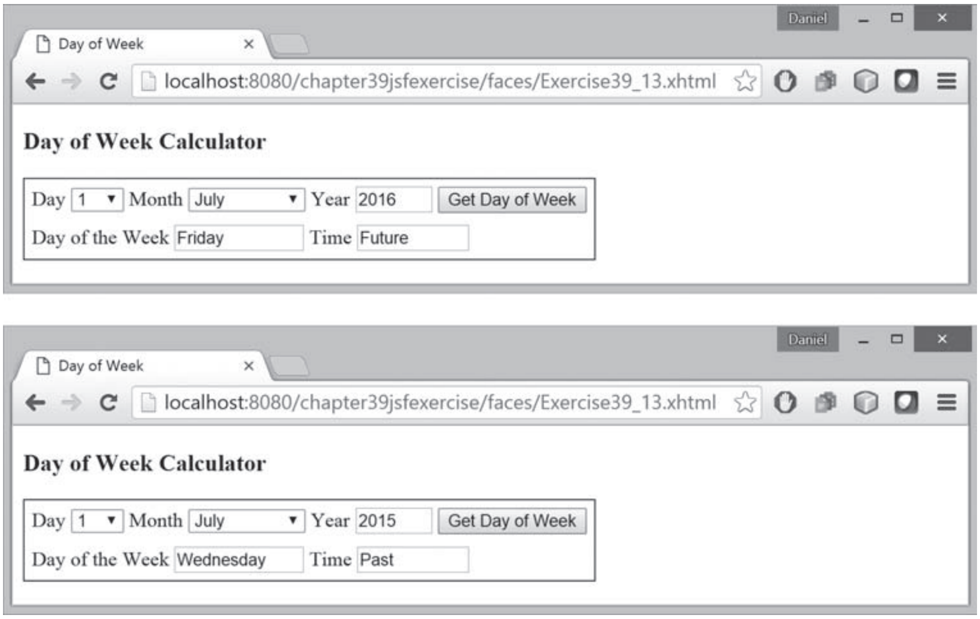
Enter four card values and click the button to determine whether the four values has a 24-point solution.

11 | 8 | 10 | 1

Find a Solution (11+1)*(10-8) is 24

FIGURE 39.38 The user enters four numbers and the program finds a solution.

***39.13** (*Day of week*) Write a program that displays the day of the week for a given day, month, and year, as shown in Figure 39.39. The program lets the user select a day, month, and year, and click the *Get Day of Week* button to display the day of week. The Time field displays “Future” if it is a future day or “Past” otherwise. Use the Zeller’s congruence to find the day of the week (see Programming Exercise 3.21).


Day of Week

localhost:8080/chapter39jsfexercise/faces/Exercise39_13.xhtml

Day of Week Calculator

Day 1 Month July Year 2016 Get Day of Week

Day of the Week Friday Time Future

Day of Week

localhost:8080/chapter39jsfexercise/faces/Exercise39_13.xhtml

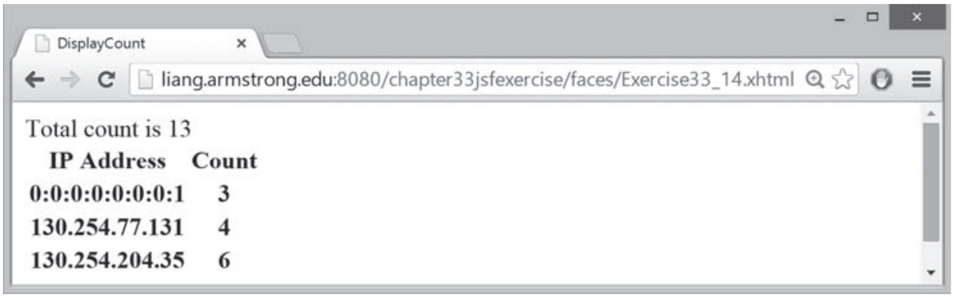
Day of Week Calculator

Day 1 Month July Year 2015 Get Day of Week

Day of the Week Wednesday Time Past

FIGURE 39.39 The user enters a day, month, and year and the program displays the day of the week.

***39.14** (*Display total count*) Revise Listing 39.22, DisplayCount.xhtml to display the total count of the button clicks form all clients and display the client's IP address and counts in increasing order of the counts, as shown in Figure 39.40.


DisplayCount
liang.armstrong.edu:8080/chapter33jsfexercise/faces/Exercise33_14.xhtml

Total count is 13

IP Address	Count
0:0:0:0:0:0:0:1	3
130.254.77.131	4
130.254.204.35	6

FIGURE 39.40 The total counts and individual client counts are displayed.

CHAPTER

40

Remote Method Invocation

Objectives

- To explain how RMI works (§40.2).
- To describe the process of developing RMI applications (§40.3).
- To distinguish between RMI and socket-level programming (§40.4).
- To develop three-tier applications using RMI (§40.5).
- To use callbacks to develop interactive applications (§40.6).

40.1 Introduction

Key Point

Remote Method Invocation is a high-level Java API for Java network programming.

Remote Method Invocation (RMI) provides a framework for building distributed Java systems. Using RMI, a Java object on one system can invoke a method in an object on another system on the network. A *distributed Java system* can be defined as a collection of cooperative distributed objects on the network. In this chapter, you will learn how to use RMI to create useful distributed applications.

40.2 RMI Basics

Key Point

RMI enables you to access a remote object and invoke its methods.

RMI is the Java Distributed Object Model for facilitating communications among distributed objects. RMI is a high-level API built on top of sockets. Socket-level programming allows you to pass data through sockets among computers. RMI enables you also to invoke methods in a remote object. Remote objects can be manipulated as if they were residing on the local host. The transmission of data among different machines is handled by the JVM transparently.

In many ways, RMI is an evolution of the client/server architecture. A *client* is a component that issues requests for services, and a *server* is a component that delivers the requested services. Like the client/server architecture, RMI maintains the notion of clients and servers, but the RMI approach is more flexible.

- An RMI component can act as both a client and a server, depending on the scenario in question.
- An RMI system can pass functionality from a server to a client, and vice versa. Typically a client/server system only passes data back and forth between server and client.

40.2.1 How Does RMI Work?

All the objects you have used before this chapter are called *local objects*. *Local objects* are accessible only within the local host. Objects that are accessible from a remote host are called *remote objects*. For an object to be invoked remotely, it must be defined in a Java interface accessible to both the server and the client. Furthermore, the interface must extend the `java.rmi.Remote` interface. Like the `java.io.Serializable` interface, `java.rmi.Remote` is a marker interface that contains no constants or methods. It is used only to identify remote objects.

The key components of the RMI architecture are listed below (see Figure 40.1):

- **Server object interface:** A subinterface of `java.rmi.Remote` that defines the methods for the server object.
- **Server class:** A class that implements the remote object interface.
- **Server object:** An instance of the server class.
- **RMI registry:** A utility that registers remote objects and provides naming services for locating objects.
- **Client program:** A program that invokes the methods in the remote server object.
- **Server stub:** An object that resides on the client host and serves as a surrogate for the remote server object.
- **Server skeleton:** An object that resides on the server host and communicates with the stub and the actual server object.

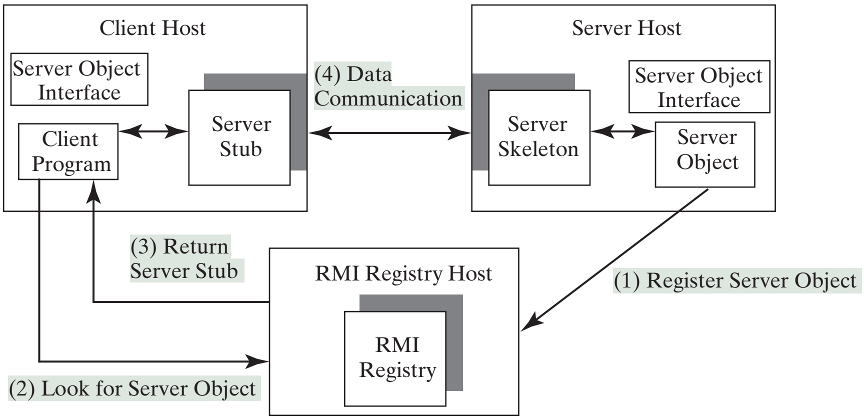


FIGURE 40.1 Java RMI uses a registry to provide naming services for remote objects, and uses the stub and the skeleton to facilitate communications between client and server.

RMI works as follows:

1. A server object is registered with the RMI registry.
2. A client looks through the RMI registry for the remote object.
3. Once the remote object is located, its stub is returned in the client.
4. The remote object can be used in the same way as a local object. Communication between the client and the server is handled through the stub and the skeleton.

The implementation of the RMI architecture is complex, but the good news is that RMI provides a mechanism that liberates you from writing the tedious code for handling parameter passing and invoking remote methods. The basic idea is to use two helper classes known as the *stub* and the *skeleton* for handling communications between client and server.

The stub and the skeleton are automatically generated. The stub resides on the client machine. It contains all the reference information the client needs to know about the server object. When a client invokes a method on a server object, it actually invokes a method that is encapsulated in the stub. The stub is responsible for sending parameters to the server and for receiving the result from the server and returning it to the client.

The skeleton communicates with the stub on the server side. The skeleton receives parameters from the client, passes them to the server for execution, and returns the result to the stub.

40.2.2 Passing Parameters

When a client invokes a remote method with parameters, passing the parameters is handled by the stub and the skeleton. Obviously, invoking methods in a remote object on a server is very different from invoking methods in a local object on a client, since the remote object is in a different address space on a separate machine. Let us consider three types of parameters:

- **Primitive data types**, such as `char`, `int`, `double`, or `boolean`, are passed by value like a local call.
- **Local object types**, such as `java.lang.String`, are also passed by value, but this is completely different from passing an object parameter in a local call. In a local call, an object parameter's reference is passed, which corresponds to the memory address of the object. In a remote call, there is no way to pass the object reference, because the address on one machine is meaningless to a different JVM. Any object can be used as

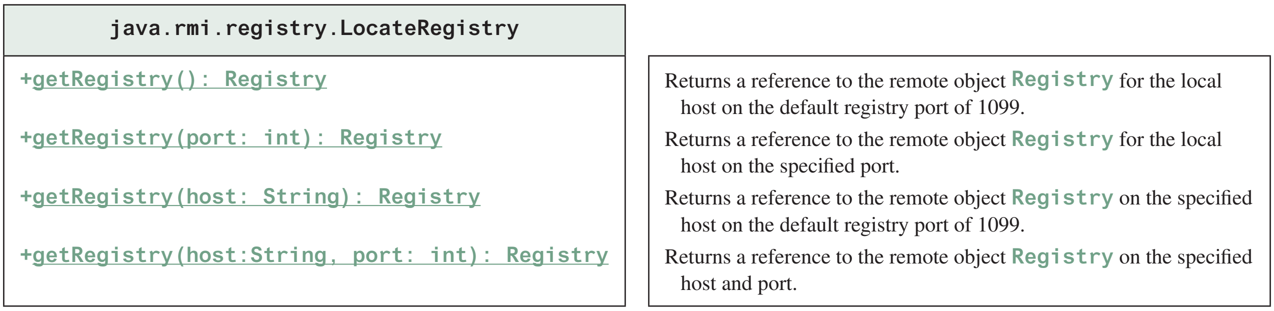
a parameter in a remote call as long as it is serializable. The stub serializes the object parameter and sends it in a stream across the network. The skeleton deserializes the stream into an object.

- **Remote object types** are passed differently from local objects. When a client invokes a remote method with a parameter of a remote object type, the stub of the remote object is passed. The server receives the stub and manipulates the parameter through it. Passing remote objects will be discussed in Section 40.6, RMI Callbacks.

40.2.3 RMI Registry

How does a client locate the remote object? The RMI registry provides the registry services for the server to register the object and for the client to locate the object.

You can use several overloaded static `getRegistry()` methods in the `LocateRegistry` class to return a reference to a `Registry`, as shown in Figure 40.2. Once a `Registry` is obtained, you can bind an object with a unique name in the registry using the `bind` or `rebind` method, or locate an object using the lookup method, as shown in Figure 40.3.



`java.rmi.registry.LocateRegistry`

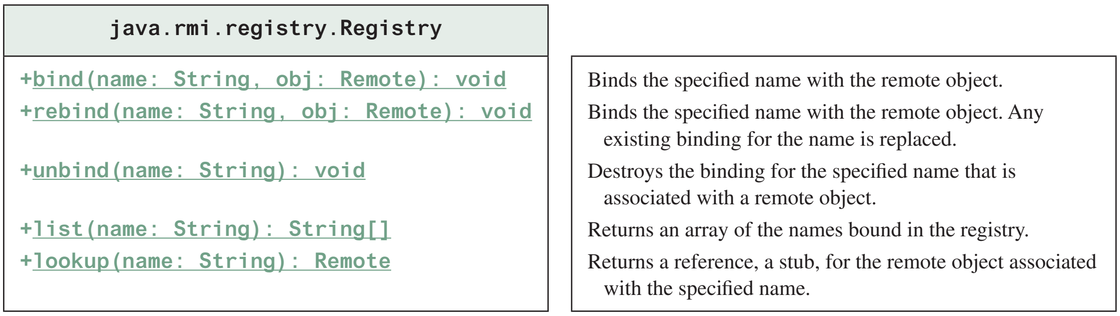
`+getRegistry(): Registry` — Returns a reference to the remote object `Registry` for the local host on the default registry port of 1099.

`+getRegistry(port: int): Registry` — Returns a reference to the remote object `Registry` for the local host on the specified port.

`+getRegistry(host: String): Registry` — Returns a reference to the remote object `Registry` on the specified host on the default registry port of 1099.

`+getRegistry(host:String, port: int): Registry` — Returns a reference to the remote object `Registry` on the specified host and port.

FIGURE 40.2 The `LocateRegistry` class provides the methods for obtaining a registry on a host.



`java.rmi.registry.Registry`

`+bind(name: String, obj: Remote): void` — Binds the specified name with the remote object.

`+rebind(name: String, obj: Remote): void` — Binds the specified name with the remote object. Any existing binding for the name is replaced.

`+unbind(name: String): void` — Destroys the binding for the specified name that is associated with a remote object.

`+list(name: String): String[]` — Returns an array of the names bound in the registry.

`+lookup(name: String): Remote` — Returns a reference, a stub, for the remote object associated with the specified name.

FIGURE 40.3 The `Registry` class provides the methods for binding and obtaining references to remote objects in a remote object registry.

40.3 Developing RMI Applications

Key Point

An RMI application consists of defining server object interface, defining a server object interface implementation class, creating and registering a server object, and developing a client program.

Now that you have a basic understanding of RMI, you are ready to write simple RMI applications. The steps in developing an RMI application are shown in Figure 40.4 and listed below.

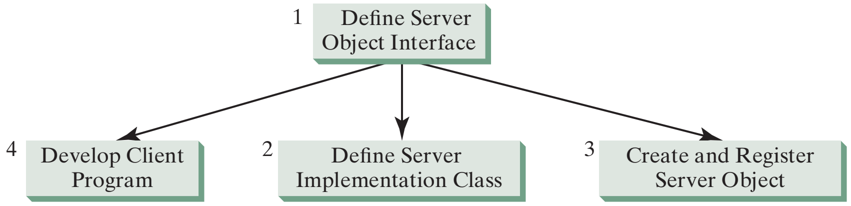

1 Define Server Object Interface
4 Develop Client Program
2 Define Server Implementation Class
3 Create and Register Server Object

FIGURE 40.4 The steps in developing an RMI application.

1. *Define a server object interface* that serves as the contract between the server and its clients, as shown in the following outline:

   ```
   public interface ServerInterface extends Remote {
     public void service1(...) throws RemoteException;
     // Other methods
   }
   ```

   A server object interface must extend the `java.rmi.Remote` interface.

2. *Define a class that implements the server object interface*, as shown in the following outline:

   ```
   public class ServerInterfaceImpl extends UnicastRemoteObject
       implements ServerInterface {
     public void service1(...) throws RemoteException {
       // Implement it
     }
     // Implement other methods
   }
   ```

   The server implementation class must extend the `java.rmi.server.UnicastRemoteObject` class. The `UnicastRemoteObject` class provides support for point-to-point active object references using TCP streams.

3. *Create a server object* from the server implementation class and register it with an RMI registry:

   ```
   ServerInterface server = new ServerInterfaceImpl(...);
   Registry registry = LocateRegistry.getRegistry();
   registry.rebind("RemoteObjectName", server);
   ```

4. *Develop a client* that locates a remote object and invokes its methods, as shown in the following outline:

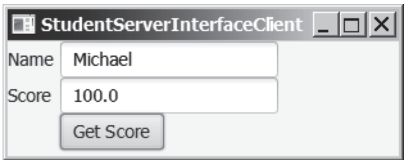
   ```
   Registry registry = LocateRegistry.getRegistry(host);
   ServerInterface server = (ServerInterfaceImpl)
     registry.lookup("RemoteObjectName");
   server.service1(...);
   ```

The example that follows demonstrates the development of an RMI application through these steps.

40.3.1 Example: Retrieving Student Scores from an RMI Server

This example creates a client that retrieves student scores from an RMI server. The client, shown in Figure 40.5, displays the score for the specified name.

1. Create a server interface named `StudentServerInterface` in Listing 40.1. The interface tells the client how to invoke the server's `findScore` method to retrieve a student score.



Name: Michael
Score: 100.0

FIGURE 40.5 You can get the score by entering a student name and clicking the *Get Score* button.

LISTING 40.1 StudentServerInterface.java

```
 1  import java.rmi.*;
 2
 3  public interface StudentServerInterface extends Remote {
 4    /**
 5     * Return the score for the specified name
 6     * @param name the student name
 7     * @return a double score or –1 if the student is not found
 8     */
 9    public double findScore(String name) throws RemoteException;
10  }
```

Any object that can be used remotely must be defined in an interface that extends the `java.rmi.Remote` interface (line 3). `StudentServerInterface`, extending `Remote`, defines the `findScore` method that can be remotely invoked by a client to find a student's score. Each method in this interface must declare that it may throw a `java.rmi.RemoteException` (line 9). Therefore, your client code that invokes this method must be prepared to catch this exception in a try-catch block.

2. Create a server implementation named `StudentServerInterfaceImpl` (Listing 40.2) that implements `StudentServerInterface`. The `findScore` method returns the score for a specified student. It returns `-1` if the score is not found.

LISTING 40.2 StudentServerInterfaceImpl.java

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class StudentServerInterfaceImpl
  extends UnicastRemoteObject
  implements StudentServerInterface {
  // Stores scores in a map indexed by name
  private HashMap<String, Double> scores =
    new HashMap<String, Double>();

  public StudentServerInterfaceImpl() throws RemoteException {
    initializeStudent();
  }

  /** Initialize student information */
  protected void initializeStudent() {
    scores.put("John", new Double(90.5));
    scores.put("Michael", new Double(100));
    scores.put("Michelle", new Double(98.5));
  }

  /** Implement the findScore method from the
   * Student interface */
  public double findScore(String name) throws RemoteException {
    Double d = (Double)scores.get(name);
27
28      if (d == null) {
29        System.out.println("Student " + name + " is not found ");
30        return -1;
31      }
32      else {
33        System.out.println("Student " + name + "\'s score is "
34          + d.doubleValue());
35        return d.doubleValue();
36      }
37    }
38  }
```

The StudentServerInterfaceImpl class implements StudentServerInterface. This class must also extend the java.rmi.server.RemoteServer class or its subclass. RemoteServer is an abstract class that defines the methods needed to create and export remote objects. Often its subclass java.rmi.server.UnicastRemoteObject is used (line 6). This subclass implements all the abstract methods defined in RemoteServer.

StudentServerInterfaceImpl implements the findScore method (lines 25–37) defined in StudentServerInterface. For simplicity, three students, John, Michael, and Michelle, and their corresponding scores are stored in an instance of java.util.HashMap named scores. HashMap is a concrete class of the Map interface in the Java Collections Framework, which makes it possible to search and retrieve a value using a key. Both values and keys are of Object type. The findScore method returns the score if the name is in the hash map, and returns -1 if the name is not found.

3. Create a server object from the server implementation and register it with the RMI server (see Listing 40.3).

Listing 40.3 RegisterWithRMIServer.java

```
1   import java.rmi.registry.*;
2
3   public class RegisterWithRMIServer {
4     /** Main method */
5     public static void main(String[] args) {
6       try {
7         StudentServerInterface obj =
8           new StudentServerInterfaceImpl();
9         Registry registry = LocateRegistry.getRegistry();
10        registry.rebind("StudentServerInterfaceImpl", obj);
11        System.out.println("Student server " + obj + " registered");
12      }
13      catch (Exception ex) {
14        ex.printStackTrace();
15      }
16    }
17  }
```

RegisterWithRMIServer contains a main method, which is responsible for starting the server. It performs the following tasks: (1) create a server object (line 8); (2) obtain a reference to the RMI registry (line 9), and (3) register the object in the registry (line 10).

4. Create a client named StudentServerInterfaceClient in Listing 40.4. The client locates the server object from the RMI registry and uses it to find the scores.

Listing 40.4 StudentServerInterfaceClient.java

```
1   import javafx.application.Application;
2   import javafx.scene.Scene;
```

```
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class StudentServerInterfaceClient extends Application {
  // Declare a Student instance
  private StudentServerInterface student;

  private Button btGetScore = new Button("Get Score");
  private TextField tfName = new TextField();
  private TextField tfScore = new TextField();

  public void start(Stage primaryStage) {
    GridPane gridPane = new GridPane();
    gridPane.setHgap(5);
    gridPane.add(new Label("Name"), 0, 0);
    gridPane.add(new Label("Score"), 0, 1);
    gridPane.add(tfName, 1, 0);
    gridPane.add(tfScore, 1, 1);
    gridPane.add(btGetScore, 1, 2);

    // Create a scene and place the pane in the stage
    Scene scene = new Scene(gridPane, 250, 250);
    primaryStage.setTitle("StudentServerInterfaceClient");
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    initializeRMI();
    btGetScore.setOnAction(e - > getScore());
  }

  private void getScore() {
    try {
      // Get student score
      double score = student.findScore(tfName.getText().trim());

      // Display the result
      if (score < 0)
        tfScore.setText("Not found");
      else
        tfScore.setText(new Double(score).toString());
    }
    catch(Exception ex) {
      ex.printStackTrace();
    }
  }

  /** Initialize RMI */
  protected void initializeRMI() {
    String host = "";

    try {
      Registry registry = LocateRegistry.getRegistry(host);
      student = (StudentServerInterface)
        registry.lookup("StudentServerInterfaceImpl");
      System.out.println("Server object " + student + " found");
    }
    catch(Exception ex) {
```

```
65        System.out.println(ex);
66      }
67    }
68
69    /**
70     * The main method is only needed for the IDE with limited
71     * JavaFX support. Not needed for running from the command line.
72     */
73    public static void main(String[] args) {
74      launch(args);
75    }
76  }
```

StudentServerInterfaceClient invokes the findScore method on the server to find the score for a specified student. The key method in StudentServerInterfaceClient is the initializeRMI method (lines 55–67), which is responsible for locating the server stub.

The lookup(String name) method (line 61) returns the remote object with the specified name. Once a remote object is found, it can be used just like a local object. The stub and the skeleton are used behind the scenes to make the remote method invocation work.

5. Follow the steps below to run this example.

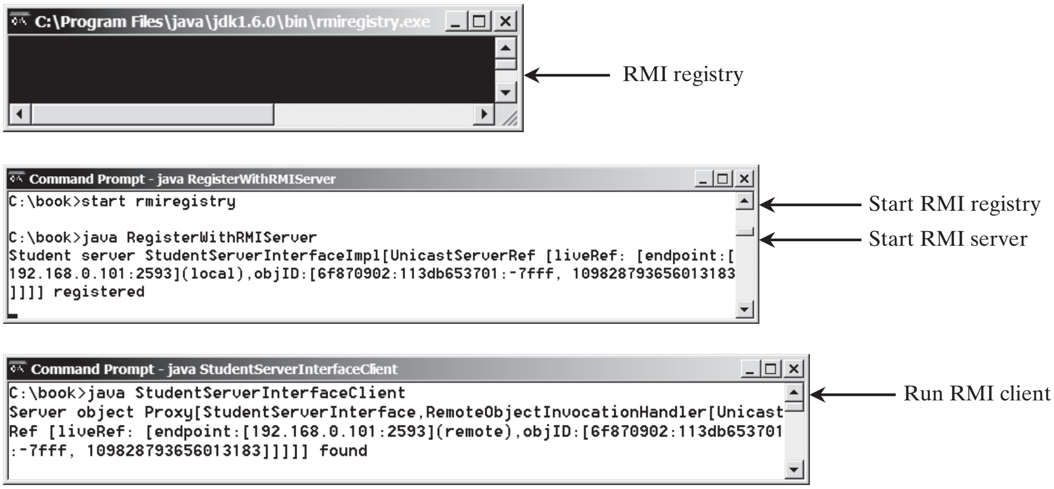
   5.1. Start the RMI registry by typing "start rmiregistry" at a DOS prompt from the book directory. By default, the port number 1099 is used by rmiregistry. To use a different port number, simply type the command "start rmiregistry *portnumber*" at a DOS prompt.

   5.2. Start the server RegisterWithRMIServer using the following command at C:\ book directory:

   C:\ book>**java RegisterWithRMIServer**

   5.3. Run the client StudentServerInterfaceClient as an application. A sample run of the application is shown in Figure 40.5(b).

Note:
You must start rmiregistry from the directory where you will run the RMI server, as shown in Figure 40.6. Otherwise, you will receive the error ClassNotFoundException on StudentServerInterfaceImpl_Stub.


C:\Program Files\java\jdk1.6.0\bin\rmiregistry.exe
RMI registry
Command Prompt - java RegisterWithRMIServer
Start RMI registry
Start RMI server
Command Prompt - java StudentServerInterfaceClient
Run RMI client

```
C:\book>start rmiregistry

C:\book>java RegisterWithRMIServer
Student server StudentServerInterfaceImpl[UnicastServerRef [liveRef: [endpoint:[
192.168.0.101:2593](local),objID:[6f870902:113db653701:-7fff, 109828793656013183
]]]] registered
```

```
C:\book>java StudentServerInterfaceClient
Server object Proxy[StudentServerInterface,RemoteObjectInvocationHandler[Unicast
Ref [liveRef: [endpoint:[192.168.0.101:2593](remote),objID:[6f870902:113db653701
:-7fff, 109828793656013183]]]]] found
```

FIGURE 40.6 To run an RMI program, first start the rmiregistry, then register the server object with the registry. The client locates it from the registry.

Note:
Server, registry, and client can be on three different machines. If you run the client and the server on separate machines, you need to place `StudentServerInterface` on both machines.

Caution:
If you modify the remote object implementation class, you need to restart the server class to reload the object to the RMI registry. In some old versions of rmiregistry, you may have to restart rmiregistry.

Check Point

40.3.1 How do you define an interface for a remote object?

40.3.2 Describe the roles of the stub and the skeleton.

40.3.3 What is `java.rmi.Remote`? How do you define a server class?

40.3.4 What is an RMI registry for? How do you create an RMI registry?

40.3.5 What is the command to start an RMI registry?

40.3.6 How do you register a remote object with the RMI registry?

40.3.7 What is the command to start a custom RMI server?

40.3.8 How does a client locate a remote object stub through an RMI registry?

40.3.9 How do you obtain a registry? How do you register a remote object? How do you locate remote object?

40.4 RMI vs. Socket-Level Programming

Key Point

RMI is a high-level network programming and socket-level network programming is low-low-level.

RMI enables you to program at a higher level of abstraction. It hides the details of socket server, socket, connection, and sending or receiving data. It even implements a multithreading server under the hood, whereas with socket-level programming, you have to explicitly implement threads for handling multiple clients.

RMI applications are scalable and easy to maintain. You can change the RMI server or move it to another machine without modifying the client program except for resetting the URL to locate the server. (To avoid resetting the URL, you can modify the client to pass the URL as a command-line parameter.) In socket-level programming, a client operation to send data requires a server operation to read it. The implementation of client and server at the socket level is tightly synchronized.

RMI clients can directly invoke the server method, whereas socket-level programming is limited to passing values. Socket-level programming is very primitive. Avoid using it to develop client/server applications. As an analogy, socket-level programming is similar to programming in assembly language, whereas RMI programming is like programming in a high-level language.

Check Point

40.10 What are the advantages of RMI over socket-level programming?

40.5 Developing Three-Tier Applications Using RMI

Key Point

RMI can be used in the middle between a client and a database to develop scalable and flexible business applications.

Three-tier applications have gained considerable attention in recent years, largely because of the demand for more scalable and load-balanced systems to replace traditional two-tier client/server database systems. A centralized database system does not just handle data access, but it also processes the business rules on data. Thus, a centralized database is usually heavily loaded, because it requires extensive data manipulation and processing. In some situations, data processing is handled by the client and business rules are stored on the client side. It is preferable to use a middle tier as a buffer between client and database. The middle tier can be used to apply business logic and rules, and to process data to reduce the load on the database.

A three-tier architecture does more than just reduce the processing load on the server. It also provides access to multiple network sites. This is especially useful to Java clients that need to access multiple databases on different servers, since the server may change.

To demonstrate, let us rewrite the example in Section 40.3.1, Example: Retrieving Student Scores from an RMI Server, to find scores stored in a database rather than a hash map. In addition, the system is capable of blocking a client from accessing a student who has not given the university permission to publish his/her score. An RMI component is developed to serve as a middle tier between client and database; it sends a search request to the database, processes the result, and returns an appropriate value to the client.

For simplicity, this example reuses the `StudentServerInterface` interface and `StudentServerInterfaceClient` class from Section 40.3.1 with no modifications. All you have to do is to provide a new implementation for the server interface and create a program to register the server with the RMI. Here are the steps to complete the program:

1. Store the scores in a database table named `Score` that contains three columns: `name`, `score`, and `permission`. The permission value is `1` or `0`, which indicates whether the student has given the university permission to release his/her grade. The following is the statement to create the table and insert three records:

```
create table Scores (name varchar(20),
  score number, permission number);

insert into Scores values ('John', 90.5, 1);
insert into Scores values ('Michael', 100, 1);
insert into Scores values ('Michelle', 100, 0);
```

2. Create a new server implementation named `Student3TierImpl` in Listing 40.5. The server retrieves a record from the `Scores` table, processes the retrieved information, and sends the result back to the client.

LISTING 40.5 Student3TierImpl.java

```
import java.rmi.*;
import java.rmi.server.*;
import java.sql.*;

public class Student3TierImpl extends UnicastRemoteObject
    implements StudentServerInterface {
  // Use prepared statement for querying DB
  private PreparedStatement pstmt;

  /** Constructs Student3TierImpl object and exports it on
  * default port.
  */
  public Student3TierImpl() throws RemoteException {
    initializeDB();
  }

  /** Constructs Student3TierImpl object and exports it on
   * specified port.
   * @param port The port for exporting
   */
  public Student3TierImpl(int port) throws RemoteException {
    super(port);
    initializeDB();
  }

  /** Load JDBC driver, establish connection and
   * create statement */
  protected void initializeDB() {
    try {
      // Load the JDBC driver
      // Class.forName("oracle.jdbc.driver.OracleDriver");
      Class.forName("com.mysql.jdbc.Driver ");

      System.out.println("Driver registered");

      // Establish connection
      /*Connection conn = DriverManager.getConnection
        ("jdbc:oracle:thin:@drake.armstrong.edu:1521:orcl",
          "scott", "tiger"); */
      Connection conn = DriverManager.getConnection
        ("jdbc:mysql://localhost/javabook", "scott", "tiger");
      System.out.println("Database connected");

      // Create a prepared statement for querying DB
      pstmt = conn.prepareStatement(
        "select * from Scores where name = ?");
    }
    catch (Exception ex) {
      System.out.println(ex);
    }
  }

  /** Return the score for specified the name
   * Return -1 if score is not found.
   */
  public double findScore(String name) throws RemoteException {
    double score = -1;
    try {
      // Set the specified name in the prepared statement
      pstmt.setString(1, name);

      // Execute the prepared statement
      ResultSet rs = pstmt.executeQuery();

      // Retrieve the score
      if (rs.next()) {
        if (rs.getBoolean(3))
          score = rs.getDouble(2);
      }
    }
    catch (SQLException ex) {
      System.out.println(ex);
    }

    return score;
  }
}
```

`Student3TierImpl` is similar to Listing 40.2, StudentServerInterfaceImpl.java in Section 40.3.1 except that the `Student3TierImpl` class finds the score from a JDBC data source instead from a hash map.

The table named Scores consists of three columns, name, score, and permission, where the latter indicates whether the student has given permission to show his/her score. Since SQL does not support a boolean type, permission is defined as a number whose value of 1 indicates true and of 0 indicates false.

The initializeDB() method (lines 28–51) establishes connections with the database and creates a prepared statement for processing the query.

The findScore method (lines 56–76) sets the name in the prepared statement, executes the statement, processes the result, and returns the score for a student whose permission is true.

3. Write a main method in the class RegisterStudent3TierServer (see Listing 40.6) that registers the server object using StudentServerInterfaceImpl, the same name as in Listing 40.2, so you can use StudentServerInterfaceClient, created in Section 40.3.1, to test the server.

Listing 40.6 RegisterStudent3TierServer.java

```java
import java.rmi.registry.*;

public class RegisterStudent3TierServer {
  public static void main(String[] args) {
    try {
      StudentServerInterface obj = new Student3TierImpl();
      Registry registry = LocateRegistry.getRegistry();
      registry.rebind("StudentServerInterfaceImpl", obj);
      System.out.println("Student server " + obj + " registered");
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

4. Follow the steps below to run this example.

    4.1. Start RMI registry by typing “start rmiregistry” at a DOS prompt from the book directory.

    4.2. Start the server RegisterStudent3TierServer using the following command at the C:\ book directory:

    C:\ book>java RegisterStudent3TierServer

    4.3. Run the client StudentServerInterfaceClient. A sample run is shown in Figure 40.6.

40.5.1 Describe how parameters are passed in RMI.

Check Point

40.6 RMI Callbacks

Key Point

RMI callbacks enable the server to invoke the methods on a client.

In a traditional client/server system, a client sends a request to a server, and the server processes the request and returns the result to the client. The server cannot invoke the methods on a client. One important benefit of RMI is that it supports *callbacks*, which enable the server to invoke methods on the client. With the RMI callback feature, you can develop interactive distributed applications.

In Section 33.6, Case Studies: Distributed TicTacToe Games, you developed a distributed TicTacToe game using stream socket programming. The example that follows demonstrates the use of the RMI callback feature to develop an interactive TicTacToe game.

All the examples you have seen so far in this chapter have simple behaviors that are easy to model with classes. The behavior of the TicTacToe game is somewhat complex. To create the classes to model the game, you need to study and understand it and distribute the process appropriately between client and server.

Clearly the client should be responsible for handling user interactions, and the server should coordinate with the client. Specifically, the client should register with the server, and the server can take two and only two players. Once a client makes a move, it should notify the server; the server then notifies the move to the other player. The server should determine the status of the game—that is, whether it has been won or drawn—and notify the players. The server should also coordinate the turns—that is, which client has the turn at a given time. The ideal approach for notifying a player is to invoke a method in the client that sets appropriate properties in the client or sends messages to a player. Figure 40.7 illustrates the relationship between clients and server.

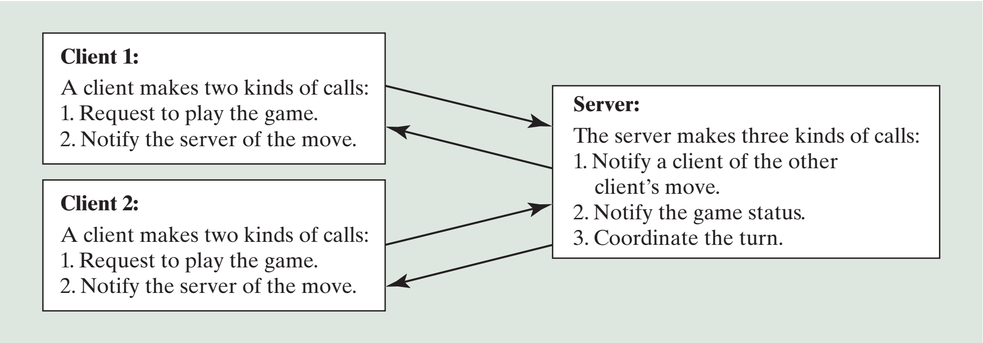

Client 1:
A client makes two kinds of calls:
1. Request to play the game.
2. Notify the server of the move.
Client 2:
A client makes two kinds of calls:
1. Request to play the game.
2. Notify the server of the move.
Server:
The server makes three kinds of calls:
1. Notify a client of the other client's move.
2. Notify the game status.
3. Coordinate the turn.

FIGURE 40.7 The server coordinates the activities with the clients.

All the calls a client makes can be encapsulated in one remote interface named TicTacToe (Listing 40.7), and all the calls the server invokes can be defined in another interface named CallBack (Listing 40.8). These two interfaces are defined as follows:

LISTING 40.7 TicTacToeInterface.java

```
import java.rmi.*;

public interface TicTacToeInterface extends Remote {
  /**
   * Connect to the TicTacToe server and return the token.
   * If the returned token is ' ', the client is not connected to
   * the server
   */
  public char connect(CallBack client) throws RemoteException;

  /** A client invokes this method to notify the server of its move*/
  public void myMove(int row, int column, char token)
    throws RemoteException;
}
```

LISTING 40.8 CallBack.java

```
import java.rmi.*;

public interface CallBack extends Remote {
  /** The server notifies the client for taking a turn */
  public void takeTurn(boolean turn) throws RemoteException;

  /** The server sends a message to be displayed by the client */
  public void notify(java.lang.String message)
    throws RemoteException;

  /** The server notifies a client of the other player's move */
  public void mark(int row, int column, char token)
    throws RemoteException;
}
```

What does a client need to do? The client interacts with the player. Assume all the cells are initially empty, and the first player takes the X token and the second player the O token. To mark a cell, the player points the mouse to the cell and clicks it. If the cell is empty, the token (X or O) is displayed. If the cell is already filled, the player's action is ignored.

From the preceding description, it is obvious that a cell is a GUI object that handles mouse-click events and displays tokens. The candidate for such an object could be a button or a panel. Panels are more flexible than buttons. The token (X or O) can be drawn on a panel in any size, but it can be displayed only as a label on a button.

Let Cell be a subclass of JPanel. You can declare a 3 × 3 grid to be an array Cell[][] cell = new Cell[3][3] for modeling the game. How do you know the state of a cell (marked or not)? You can use a property named marked of the boolean type in the Cell class. How do you know whether the player has a turn? You can use a property named myTurn of boolean. This property (initially false) can be set by the server through a callback.

The Cell class is responsible for drawing the token when an empty cell is clicked, so you need to write the code for listening to the MouseEvent and for painting the shape for tokens X and O. To determine which shape to draw, introduce a variable named marker of the char type. Since this variable is shared by all the cells in a client, it is preferable to declare it in the client and to declare the Cell class as an inner class of the client so this variable will be accessible to all the cells.

Now let us turn our attention to the server side. What does the server need to do? The server needs to implement TicTacToeInterface and notify the clients of the game status. The server has to record the moves in the cells and check the status every time a player makes a move. The status information can be kept in a 3 × 3 array of char. You can implement a method named isFull() to check whether the board is full and a method named isWon(token) to check whether a specific player has won.

Once a client is connected to the server, the server notifies the client which token to use—that is, X for the first client and O for the second. Once a client notifies the server of its move, the server checks the game status and notifies the clients.

Now the most critical question is how the server notifies a client. You know that a client invokes a server method by creating a server stub on the client side. A server cannot directly invoke a client, because the client is not declared as a remote object. The CallBack interface was created to facilitate the server's callback to the client. In the implementation of CallBack, an instance of the client is passed as a parameter in the constructor of CallBack. The client creates an instance of CallBack and passes its stub to the server, using a remote method named connect() defined in the server. The server then invokes the client's method through a CallBack instance. The triangular relationship of client, CallBack implementation, and server is shown in Figure 40.8.

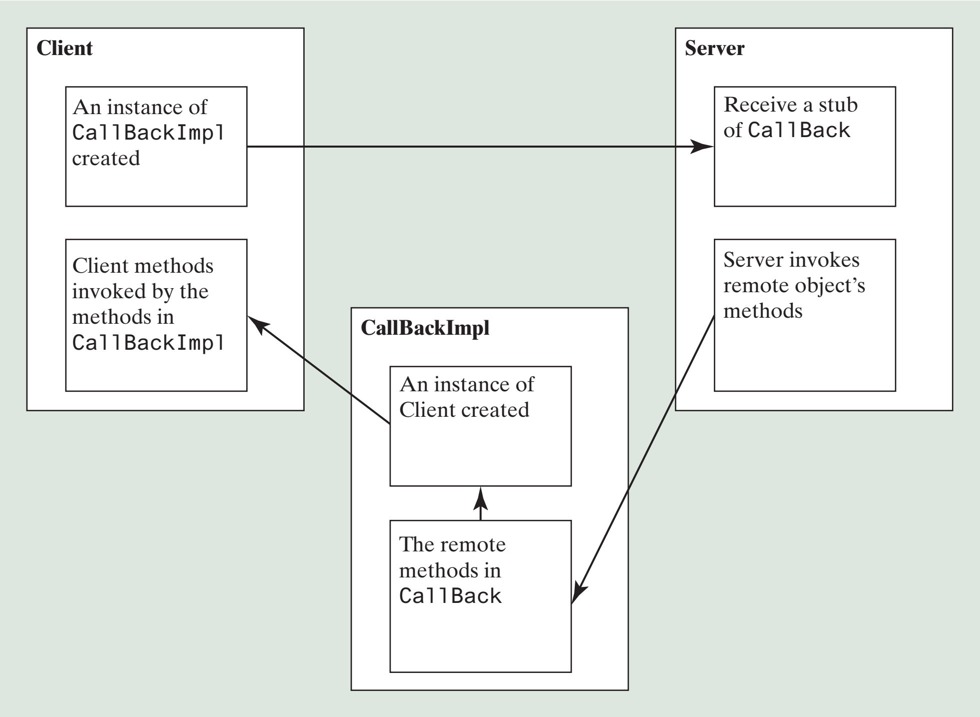

Client
An instance of CallBackImpl created
Client methods invoked by the methods in CallBackImpl
Server
Receive a stub of CallBack
Server invokes remote object's methods
CallBackImpl
An instance of Client created
The remote methods in CallBack

FIGURE 40.8 The server receives a CallBack stub from the client and invokes the remote methods defined in the CallBack interface, which can invoke the methods defined in the client.

Here are the steps to complete the example.

1. Create TicTacToeImpl.java (Listing 40.9) to implement TicTacToeInterface. Add a main method in the program to register the server with the RMI.

LISTING 40.9 TicTacToeImpl.java

```java
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.rmi.registry.*;

public class TicTacToeImpl extends UnicastRemoteObject
    implements TicTacToeInterface {
  // Declare two players, used to call players back
  private CallBack player1 = null;
  private CallBack player2 = null;

  // board records players' moves
  private char[][] board = new char[3][3];

  /** Constructs TicTacToeImpl object and
    exports it on default port.
   */
  public TicTacToeImpl() throws RemoteException {
    super();
  }

  /** Constructs TicTacToeImpl object and exports it on specified
   * port.
   * @param port The port for exporting
   */
  public TicTacToeImpl(int port) throws RemoteException {
    super(port);
  }

  /**
   * Connect to the TicTacToe server and return the token.
   * If the returned token is ' ', the client is not connected to
   * the server
   */
  public char connect(CallBack client) throws RemoteException {
    if (player1 == null) {
      // player1 (first player) registered
      player1 = client;
      player1.notify("Wait for a second player to join");
      return 'X';
    }
    else if (player2 == null) {
      // player2 (second player) registered
      player2 = client;
      player2.notify("Wait for the first player to move");
      player2.takeTurn(false);
      player1.notify("It is my turn (X token)");
      player1.takeTurn(true);
      return 'O';
    }
    else {
      // Already two players
      client.notify("Two players are already in the game");
      return ' ';
    }
  }

  /** A client invokes this method to notify the
    server of its move*/
  public void myMove(int row, int column, char token)
      throws RemoteException {
    // Set token to the specified cell
    board[row][column] = token;

    // Notify the other player of the move
    if (token == 'X')
      player2.mark(row, column, 'X');
    else
      player1.mark(row, column, 'O');

    // Check if the player with this token wins
    if (isWon(token)) {
      if (token == 'X') {
        player1.notify("I won!");
        player2.notify("I lost!");
        player1.takeTurn(false);
      }
      else {
        player2.notify("I won!");
        player1.notify("I lost!");
        player2.takeTurn(false);
      }
    }
    else if (isFull()) {
      player1.notify("Draw!");
      player2.notify("Draw!");
    }
    else if (token == 'X') {
      player1.notify("Wait for the second player to move");
      player1.takeTurn(false);
      player2.notify("It is my turn, (O token)");
      player2.takeTurn(true);
    }
    else if (token == 'O') {
      player2.notify("Wait for the first player to move");
      player2.takeTurn(false);
      player1.notify("It is my turn, (X token)");
      player1.takeTurn(true);
    }
  }

  /** Check if a player with the specified token wins */
  public boolean isWon(char token) {
    for (int i = 0; i < 3; i++)
      if ((board[i][0] == token) && (board[i][1] == token)
        && (board[i][2] == token))
        return true;

    for (int j = 0; j < 3; j++)
      if ((board[0][j] == token) && (board[1][j] == token)
        && (board[2][j] == token))
        return true;

    if ((board[0][0] == token) && (board[1][1] == token)
      && (board[2][2] == token))
      return true;

    if ((board[0][2] == token) && (board[1][1] == token)
      && (board[2][0] == token))
      return true;

    return false;
  }

  /** Check if the board is full */
  public boolean isFull() {
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
        if (board[i][j] == '\u0000')
          return false;

    return true;
  }

  public static void main(String[] args) {
    try {
      TicTacToeInterface obj = new TicTacToeImpl();
      Registry registry = LocateRegistry.getRegistry();
      registry.rebind("TicTacToeImpl", obj);
      System.out.println("Server " + obj + " registered");
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

2. Create CallBackImpl.java (Listing 40.10) to implement the CallBack interface.

Listing 40.10 CallBackImpl.java

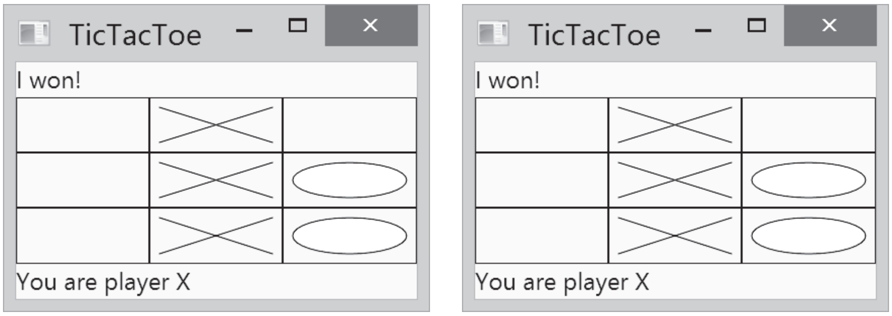
```java
import java.rmi.*;
import java.rmi.server.*;

public class CallBackImpl extends UnicastRemoteObject
    implements CallBack {
  // The client will be called by the server through callback
  private TicTacToeClientRMI thisClient;

  /** Constructor */
  public CallBackImpl(Object client) throws RemoteException {
    thisClient = (TicTacToeClientRMI)client;
  }

  /** The server notifies the client for taking a turn */
  public void takeTurn(boolean turn) throws RemoteException {
    thisClient.setMyTurn(turn);
  }

  /** The server sends a message to be displayed by the client */
  public void notify(String message )throws RemoteException {
    thisClient.setMessage(message);
  }

  /** The server notifies a client of the other player's move */
  public void mark(int row, int column, char token)
      throws RemoteException {
    thisClient.mark(row, column, token);
  }
}
```

3. Create a client named TicTacToeClientRMI (Listing 40.11) for interacting with a player and communicating with the server. Enable it to run standalone.

Listing 40.11 TicTacToeClientRMI.java

```java
import java.rmi.*;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Line;
import javafx.scene.shape.Ellipse;

import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;

public class TicTacToeClientRMI extends Application {
  // marker is used to indicate the token type
  private char marker;

  // myTurn indicates whether the player can move now
  private boolean myTurn = false;

  // Indicate which player has a turn, initially it is the X player
  private char whoseTurn = 'X';

  // Create and initialize cell
  private Cell[][] cell = new Cell[3][3];

  // Create and initialize a status label
  private Label lblStatus = new Label("X's turn to play");

  // ticTacToe is the game server for coordinating
  // with the players
  private TicTacToeInterface ticTacToe;

  private Label lblIdentification = new Label();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Pane to hold cell
    GridPane pane = new GridPane();
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
        pane.add(cell[i][j] = new Cell(i, j), j, i);

    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(pane);
    borderPane.setTop(lblStatus);
    borderPane.setBottom(lblIdentification);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 450, 170);
    primaryStage.setTitle("TicTacToe"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    new Thread( () -> {
    try {
      initializeRMI();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }}).start();
  }

  /** Initialize RMI */
  protected boolean initializeRMI() throws Exception {
    String host = "";

    try {
      Registry registry = LocateRegistry.getRegistry(host);
      ticTacToe = (TicTacToeInterface)
        registry.lookup("TicTacToeImpl");
      System.out.println
        ("Server object " + ticTacToe + " found");
    }
    catch (Exception ex) {
      System.out.println(ex);
    }

    // Create callback for use by the
    // server to control the client
    CallBackImpl callBackControl = new CallBackImpl(this);

    if (
      (marker =
        ticTacToe.connect((CallBack)callBackControl)) != ' ')
    {
      System.out.println("connected as " + marker + " player.");
      Platform.runLater(() ->
        lblIdentification.setText("You are player " + marker));
      return true;
    }
    else {
      System.out.println("already two players connected as ");
      return false;
    }
  }

  /** Set variable myTurn to true or false */
  public void setMyTurn(boolean myTurn) {
    this.myTurn = myTurn;
  }

  /** Set message on the status label */
  public void setMessage(String message) {
    Platform.runLater(() -> lblStatus.setText(message));
  }

  /** Mark the specified cell using the token */
  public void mark(int row, int column, char token) {
    cell[row][column].setToken(token);
  }

  // An inner class for a cell
  public class Cell extends Pane {
    // marked indicates whether the cell has been used
    private boolean marked = false;

    // row and column indicate where the cell appears on the board
    int row, column;

    // Token used for this cell
    private char token = ' ';

    public Cell(final int row, final int column) {
      this.row = row;
      this.column = column;
      setStyle("-fx-border-color: black");
      this.setPrefSize(2000, 2000);
      this.setOnMouseClicked(e -> handleMouseClick());
    }

    /** Return token */
    public char getToken() {
      return token;
    }

    /** Set a new token */
    public void setToken(char c) {
      token = c;
      marked = true;

```

```java
      if (token == 'X') {
        Line line1 = new Line(10, 10,
          this.getWidth() - 10, this.getHeight() - 10);
        line1.endXProperty().bind(this.widthProperty().subtract(10));
        line1.endYProperty().bind(this.heightProperty().subtract(10));
        Line line2 = new Line(10, this.getHeight() - 10,
          this.getWidth() - 10, 10);
        line2.startYProperty().bind(
          this.heightProperty().subtract(10));
        line2.endXProperty().bind(this.widthProperty().subtract(10));

        // Add the lines to the pane
        Platform.runLater(() ->
          this.getChildren().addAll(line1, line2));
      }
      else if (token == 'O') {
        Ellipse ellipse = new Ellipse(this.getWidth() / 2,
          this.getHeight() / 2, this.getWidth() / 2 - 10,
          this.getHeight() / 2 - 10);
        ellipse.centerXProperty().bind(
          this.widthProperty().divide(2));
        ellipse.centerYProperty().bind(
          this.heightProperty().divide(2));
        ellipse.radiusXProperty().bind(
          this.widthProperty().divide(2).subtract(10));
        ellipse.radiusYProperty().bind(
          this.heightProperty().divide(2).subtract(10));
        ellipse.setStroke(Color.BLACK);
        ellipse.setFill(Color.WHITE);

        Platform.runLater(() ->
          getChildren().add(ellipse)); // Add the ellipse to the pane
      }
    }

    /* Handle a mouse click event */
    private void handleMouseClick() {
      if (myTurn && !marked) {
        // Mark the cell
        setToken(marker);

        // Notify the server of the move
        try {
          ticTacToe.myMove(row, column, marker);
        }
        catch (RemoteException ex) {
          System.out.println(ex);
        }
      }
    }
  }

  /**
   * The main method is only needed for the IDE with limited
   * JavaFX support. Not needed for running from the command line.
   */
  public static void main(String[] args) {
    launch(args);
  }
}
```



I won!

You are player X

I won!

You are player X

FIGURE 40.9 Two players play each other through the RMI server.

4. Follow the steps below to run this example.

   4.1. Start RMI registry by typing “start rmiregistry” at a DOS prompt from the book directory.

   4.2. Start the server TicTacToeImpl using the following command at the C:\ book directory:
   C:\ book>java TicTacToeImpl

   4.3. Run the client TicTacToeClientRMI. A sample run is shown in Figure 40.9.

TicTacToeInterface defines two remote methods, connect(CallBack client) and myMove(int row, int column, char token). The connect method plays two roles: one is to pass a CallBack stub to the server, and the other is to let the server assign a token for the player. The myMove method notifies the server that the player has made a specific move.

The CallBack interface defines three remote methods, takeTurn(boolean turn), notify(String message), and mark(int row, int column, char token). The takeTurn method sets the client’s myTurn property to true or false. The notify method displays a message on the client’s status label. The mark method marks the client’s cell with the token at the specified location.

TicTacToeImpl is a server implementation for coordinating with the clients and managing the game. The variables player1 and player2 are instances of CallBack, each of which corresponds to a client, passed from a client when the client invokes the connect method. The variable board records the moves by the two players. This information is needed to determine the game status. When a client invokes the connect method, the server assigns a token X for the first player and O for the second player, and accepts only two players. You can modify the program to accept additional clients as observers. (See Exercise 40.7 for more details).

Once two players are in the game, the server coordinates the turns between them. When a client invokes the myMove method, the server records the move and notifies the other player by marking the other player’s cell. It then checks to see whether the player wins or whether the board is full. If neither condition applies and therefore the game continues, the server gives a turn to the other player.

The CallBackImpl implements the CallBack interface. It creates an instance of TicTacToeClientRMI through its constructor. The CallBackImpl relays the server request to the client by invoking the client’s methods. When the server invokes the takeTurn method, CallBackImpl invokes the client’s setMyTurn() method to set the property myTurn in the client. When the server invokes the notify() method, CallBackImpl invokes the client’s setMessage() method to set the message on the client’s status label. When the server invokes the mark method, CallBackImpl invokes the client’s mark method to mark the specified cell.

Interestingly, obtaining the TicTacToeImpl stub for the client is different from obtaining the CallBack stub for the server. The TicTacToeImpl stub is obtained by invoking the lookup() method through the RMI registry, and the CallBack stub is passed to the server through the connect method in the TicTacToeImpl stub. It is a common practice to obtain the first stub with the lookup method, but to pass the subsequent stubs as parameters through remote method invocations.

Since the variables myTurn and marker are defined in TicTacToeClientRMI, the Cell class is defined as an inner class within TicTacToeClientRMI in order to enable all the cells in the client to access them. Exercise 40.8 suggests alternative approaches that implement the Cell as a noninner class.

Check Point

40.6.1 What is the problem if the connect method in the TicTacToeInterface is defined as

```
public boolean connect(CallBack client, char token)
  throws RemoteException;
```

or as

```
public boolean connect(CallBack client, Character token)
  throws RemoteException;
```

40.6.2 What is callback? How does callback work in RMI?

Key Terms

callback 40-13	skeleton 40-3
RMI registry 40-3	stub 40-3

Chapter Summary

1. RMI is a high-level Java API for building distributed applications using distributed objects.

2. The key idea of RMI is its use of stubs and skeletons to facilitate communications between objects. The stub and skeleton are automatically generated, which relieves programmers of tedious socket-level network programming.

3. For an object to be used remotely, it must be defined in an interface that extends the java.rmi.Remote interface.

4. In an RMI application, the initial remote object must be registered with the RMI registry on the server side and be obtained using the lookup method through the registry on the client side. Subsequent uses of stubs of other remote objects may be passed as parameters through remote method invocations.

5. RMI is especially useful for developing scalable and load-balanced multitier distributed applications.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

Programming Exercises

MyProgrammingLab™

Section 40.3

***40.1** (*Limit the number of clients*) Modify the example in Section 40.3.1, Example: Retrieving Student Scores from an RMI Server, to limit the number of concurrent clients to 10.

***40.2** (*Compute loan*) Rewrite Programming Exercise 33.1 using RMI. You need to define a remote interface for computing monthly payment and total payment.

****40.3** (*Web visit count*) Rewrite Programming Exercise 33.4 using RMI. You need to define a remote interface for obtaining and increasing the count.

****40.4** (*Display and add addresses*) Rewrite Programming Exercise 33.6 using RMI. You need to define a remote interface for adding addresses and retrieving address information.

Section 40.5

****40.5** (*Address in a database table*) Rewrite Programming Exercise 40.4. Assume the address is stored in a table.

****40.6** (*Three-tier application*) Use the three-tier approach to modify Programming Exercise 40.4, as follows:

- Create a JavaFX client to manipulate student information, as shown in Figure 33.23a.
- Create a remote object interface with methods for retrieving, inserting, and updating student information, and an object implementation for the interface.

Section 40.6

****40.7** (*Chat*) Rewrite Programming Exercise 33.13 using RMI. You need to define a remote interface for sending and receiving a message.

****40.8** (*Improve TicTacToe*) Modify the TicTacToe example in Section 40.6, RMI Callbacks, as follows:

- Allow a client to connect to the server as an observer to watch the game.
- Rewrite the `Cell` class as a noninner class.

CHAPTER

41

WEB SERVICES

Objectives

- To describe what a Web service is (§41.1).
- To create a Web service class (§41.2).
- To publish and test a Web service (§41.3).
- To create a Web service client reference (§41.4).
- To explain the role of WSDL (§41.4).
- To pass arguments of object type in a Web service (§41.5).
- To discover how a client communicates with a Web service (§41.5).
- To describe what SOAP requests and SOAP responses are (§41.5).
- To track a session in Web services (§41.6).

41.1 Introduction

Key Point

Web services is about sharing objects on the Internet.

Web service is a technology that enables programs to communicate through HTTP on the Internet. Web services enable a program on one system to invoke a method in an object on another system. You can develop and use Web services using any languages on any platform. Web services are simple and easy to develop.

Web services run on the Web using HTTP. There are several APIs for Web services. A popular standard is the *Simple Object Access Protocol* (SOAP), which is based on XML. The computer on which a Web service resides is referred to as a *server*. The server needs to make the service available to the client, known as *publishing a Web service*. Using a Web service from a client is known as *consuming a Web service*.

A client interacts with a Web service through a *proxy object*. The proxy object facilitates the communication between the client and the Web service. The client passes arguments to invoke methods on the proxy object. The proxy object sends the request to the server and receives the result back from the server, as shown in Figure 41.1.

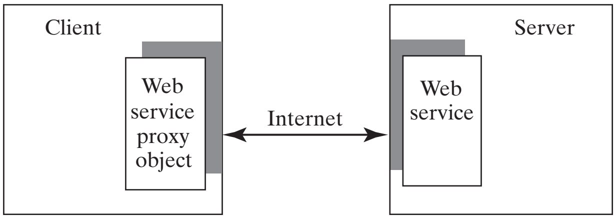


FIGURE 41.1 A proxy object serves as a facilitator between a client and a Web service.

Check Point

41.1.1 What is a Web service?

41.1.2 Can you invoke a Web service from a language other than Java?

41.1.3 Do Web services support callback? That is, can a Web service call a method from a client's program?

41.1.4 What is SOAP? What is it to publish a Web service? What is it to consume a Web service? What is the role of a proxy object?

41.2 Creating Web Services

Key Point

An IDE such as NetBeans is an effective tool for developing and deploying Web services.

There are many tools for creating Web services. This book demonstrates creating Web services using NetBeans.

Note

Apache Tomcat Server does not work well with Web services. To develop and deploy Web services using NetBeans, you need to install GlassFish. For information on how to install GlassFish on NetBeans.

We now create a Web service for obtaining student scores. A Web service is a class that contains the methods for the client to invoke. Name the class `ScoreService` with a method named `findScore(String name)` that returns the score for a student.

First, you need to create a *Web project* using the following steps:

1. Choose **`File, New Project`** to display the New Project dialog box. In the New Project dialog box, choose **`Java Web`** in the Categories pane and choose **`Web Application`** in the Projects pane. Click *Next* to display the New Web Application dialog box.

2. Enter WebServiceProject as the project name, specify the location where you want the project to be stored, and click Next to display the Server and Setting dialog.
3. Select **GlassFish 4** as the server and **Java EE 7 Web** as the Java EE version. Click **Finish** to create the project.

Now you can create the ScoreService class in the project as follows:

1. Right-click the WebServiceProject in the Project pane to display a context menu. Choose **New, Web Service** to display the New Web Service dialog box. (If you don't see **Web Service**, click **New**, **Other** to display the New File dialog box to choose Web Service in this dialog box.)
2. Enter ScoreService in the Web Service Name field and enter chapter41 in the Package field. Click *Finis*h to create ScoreService.
3. Complete the source code as shown in Listing 41.1.

LISTING 41.1 ScoreService.java

```
 1  package chapter41;
 2
 3  import java.util.HashMap;
 4  import javax.jws.WebService; // For annotation @WebService
 5  import javax.jws.WebMethod; // For annotation @WebMethod
 6
 7  @WebService(name = "ScoreService", serviceName = "ScoreWebService")
 8  public class ScoreService {
 9    // Stores scores in a map indexed by name
10    private HashMap<String, Double> scores =
11      new HashMap<String, Double>();
12
13    public ScoreService() {
14      scores.put("John", 90.5);
15      scores.put("Michael", 100.0);
16      scores.put("Michelle", 98.5);
17    }
18
19    @WebMethod(operationName = "findScore")
20    public double findScore(String name) {
21      Double d = scores.get(name);
22
23      if (d == null) {
24        System.out.println("Student " + name + " is not found ");
25        return -1;
26      }
27      else {
28        System.out.println("Student " + name + "\'s score is "
29          + d.doubleValue());
30        return d.doubleValue();
31      }
32    }
33  }
```

Lines 4–5 import the annotations used in the program in lines 7 and 19. Annotation is a new feature in Java, which enables you to simplify coding. The compiler will automatically generate the code for the annotated directives. So, it frees the programmer from writing the detailed *boilerplate code* that could be generated mechanically. The annotation (line 7)

```
@WebService(name = "ScoreService", serviceName = "ScoreWebService")
```

tells the compiler that the class ScoreService is associated with the Web service named ScoreWebService.

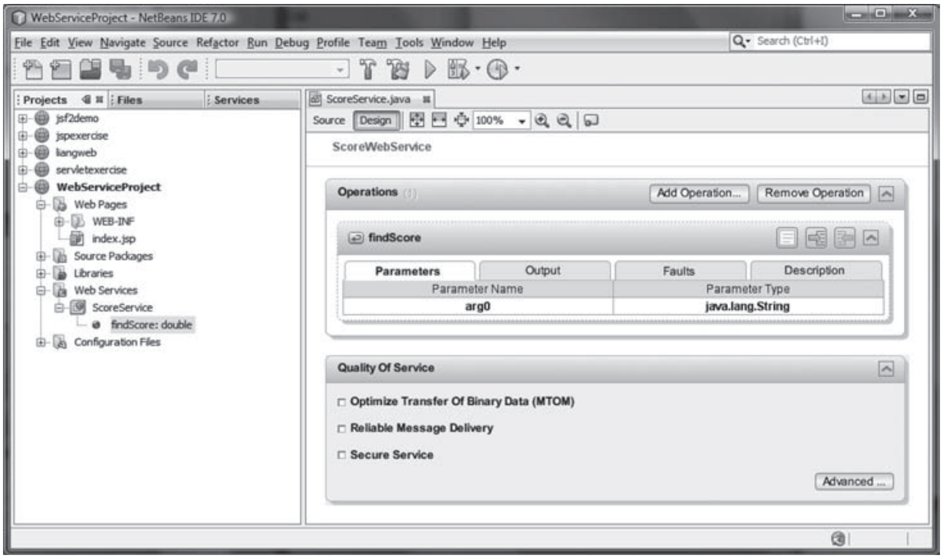
The annotation (line 19)

```
@WebMethod(operationName = "findScore")
```

indicates that `findScore` is a method that can be invoked from a client.

The `findScore` method returns a score if the name is in the hash map. Otherwise, it returns `-1.0`.

You can manually type the code for the service, or create it from the Design tab, as shown in Figure 41.2.



Parameter Name	Parameter Type
arg0	java.lang.String

□ Optimize Transfer Of Binary Data (MTOM)

□ Reliable Message Delivery

□ Secure Service

FIGURE 41.2 The services can also be created from the Design pane.

41.3 Deploying and Testing Web Services

Key Point

Deploying a Web service is to make it available on the Internet for other programs to use.

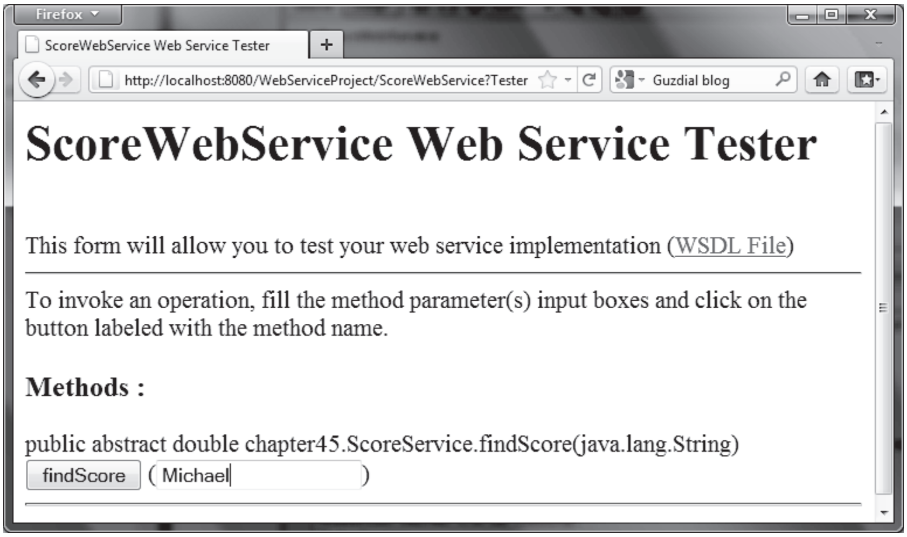
After a Web service is created, you need to deploy it for clients to use. Deploying Web services is also known as *publishing Web services*. To deploy it, right-click the `WebServiceProject` in the Project to display a context menu and choose **`Deploy`**. This command will first undeploy the service if it was deployed and then redeploy it.

Now you can test the Web service by entering the follow URL in a browser, as shown in Figure 41.3.

http://localhost:8080/WebServiceProject/ScoreWebService?Tester

Note `ScoreWebService` is the name you specified in line 7 in Listing 41.1. This Web service has only one remote method named `findScore`. You can define an unlimited number of remote methods in a Web service class. If so, all these methods will be displayed in the test page.

To test the `findScore` method, enter `Michael` and click the *findScore* button. You will see that the method returns `100.0`, as shown in Figure 41.4.



Firefox
ScoreWebService Web Service Tester
http://localhost:8080/WebServiceProject/ScoreWebService?Tester
Guzdial blog

ScoreWebService Web Service Tester

This form will allow you to test your web service implementation (WSDL File)

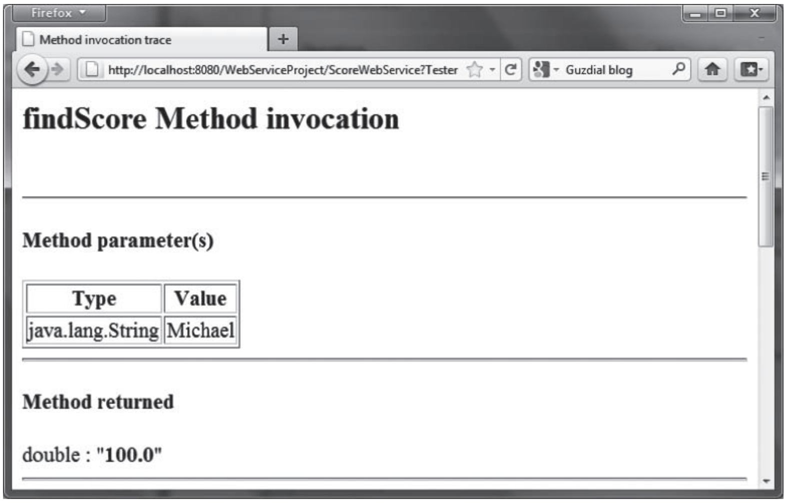
To invoke an operation, fill the method parameter(s) input boxes and click on the button labeled with the method name.

Methods :

public abstract double chapter45.ScoreService.findScore(java.lang.String)

findScore (Michael)

FIGURE 41.3 The test page enables you to test Web services.



Firefox
Method invocation trace
http://localhost:8080/WebServiceProject/ScoreWebService?Tester
Guzdial blog

findScore Method invocation

Method parameter(s)

Type	Value
java.lang.String	Michael

Method returned

double : "100.0"

FIGURE 41.4 The method returns a test value.

Note

If your computer is connected to the Internet, you can test Web services from another computer by entering the following URL:

http://*host*:8080/WebServiceProject/ScoreWebService?Tester

Where *host* is the host name or IP address of the server on which the Web service is running. On Windows, you can find your IP address by typing the command **`ipconfig`**.

Note

If you are running the server on Windows, the firewall may prevent remote clients from accessing the service. To enable it, do the following:

1. In the Windows control panel, click Windows Firewall to display the Windows Firewall dialog box.

2. In the Advanced tab, double-click Local Area Connection to display the Advanced Settings dialog box. Check *Web Server (HTTP)* to enable HTTP access to the server.

3. Click *OK* to close the dialog box.

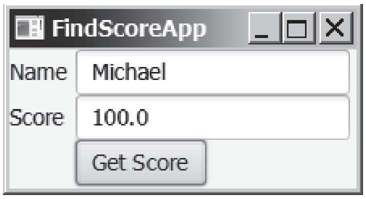
41.4 Consuming Web Services

Key Point

Consuming a Web service is for a client to use a Web service.

After a Web service is published, you can write a client program to use it. A client can be any program (standalone application, servlet/JSP/JSF application, or another Web service) and written in any language.

We will use NetBeans to create a Web service client. Our client is a GUI application. The application simply lets the user enter a name and displays the score, as shown in Figure 41.5.



Name: Michael

Score: 100.0

Figure 41.5 The client uses the Web service to find scores.

Let us create a project for the client. The project named `ScoreWebServiceClientProject` can be created as follows:

1. Choose **`File, New Project`** to display the New Project dialog box.

2. In the New Project dialog box, choose **`Java`** in the Categories pane and choose **`Java Application`** in the Projects pane. Click *Next* to display the New Java Application dialog box.

3. Enter `ScoreWebServiceClientProject` as the project name, specify the location where you want the project to be stored, and uncheck the *Create Main Class* check box. Click *Finish* to create the project.

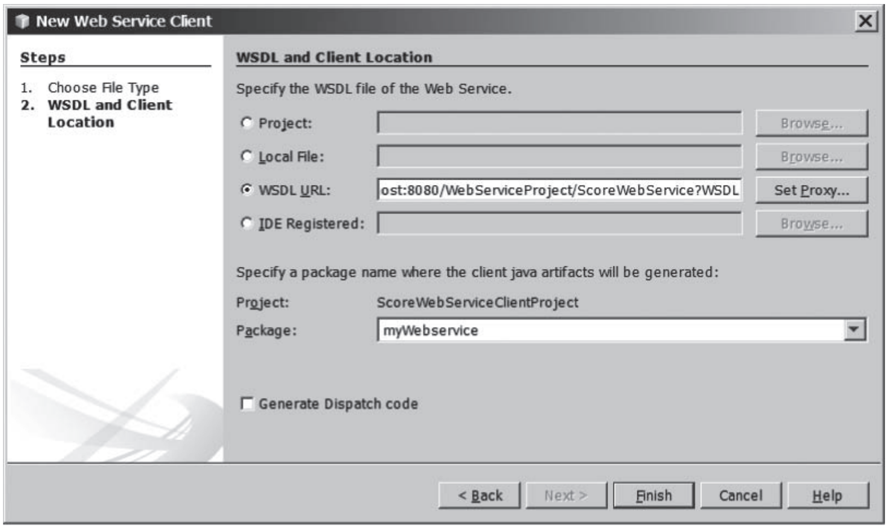
You need to create a Web service reference to this project. The reference will enable you to create a proxy object to interact with the Web service. Here are the steps to create a Web service reference:

1. Right-click the `ScoreWebServiceClientProject` in the Project pane to display a context menu. Choose **`New, Web Service Client`** to display the New Web Service Client dialog box, as shown in Figure 41.6.

2. Check the *WSDL URL* radio button and enter http://localhost:8080/WebServiceProject/ScoreWebService?WSDL in the WSDL URL field.

3. Enter `myWebservice` in the package name field. Click *Finish* to generate the Web service reference.

Now you will see `ScoreWebService` created in the Web Service References folder in the Projects tab. The IDE has generated many supporting files for the reference. You can view all the generated .java files from the Files tab in the project pane, as shown in Figure 41.7. These files will be used by the proxy object to interact with the Web service.



New Web Service Client

Steps

1. Choose File Type
2. **WSDL and Client Location**

WSDL and Client Location

Specify the WSDL file of the Web Service.

○ Project: [ ] Browse...

○ Local File: [ ] Browse...

● WSDL URL: ost:8080/WebServiceProject/ScoreWebService?WSDL Set Proxy...

○ IDE Registered: [ ] Browse...

Specify a package name where the client java artifacts will be generated:

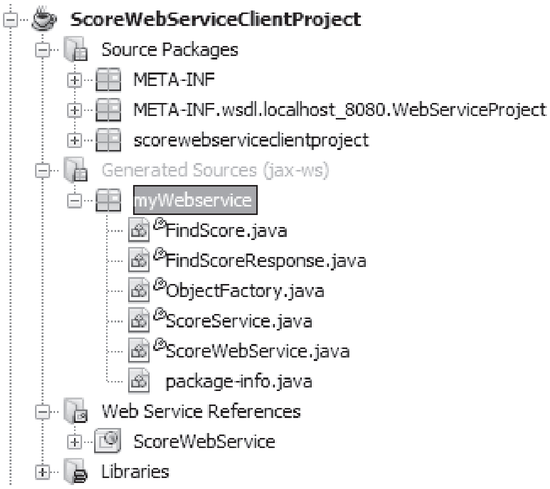
Project: ScoreWebServiceClientProject

Package: myWebservice

□ Generate Dispatch code

< Back | Next > | Finish | Cancel | Help

FIGURE 41.6 The New Web Service Client dialog box creates a Web service reference.



```
ScoreWebServiceClientProject
  Source Packages
    META-INF
    META-INF.wsdl.localhost_8080.WebServiceProject
    scorewebserviceclientproject
  Generated Sources (jax-ws)
    myWebservice
      FindScore.java
      FindScoreResponse.java
      ObjectFactory.java
      ScoreService.java
      ScoreWebService.java
      package-info.java
  Web Service References
    ScoreWebService
  Libraries
```

FIGURE 41.7 You can see the automatically generated boilerplate code for Web services in the Generated Sources folder in the client's project.

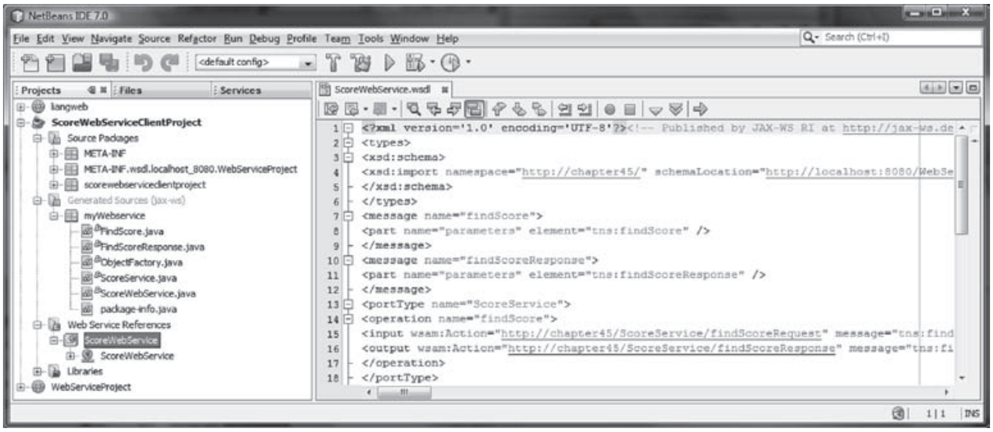
Note

When you created a Web service reference, you entered a WSDL URL, as shown in Figure 41.6. This creates a .wsdl file. In this case, it is named ScoreWebService.wsdl under the Web Service References folder, as shown in Figure 41.8. So *what is WSDL*? WSDL stands for *Web Service Description Language*. A .wsdl file is an XML file that describes the available Web service to the client—i.e., the remote methods, their parameters and return value types, and so on.

Note

If the Web service is modified, you need to refresh the reference for the client. To do so, right-click the Web service node under Web Service References to display a context menu and choose **`Refresh Client`**.

Now you are ready to create a client for the Web service. Right-click the `ScoreWebServiceClientProject` node in the Project pane to display a context menu, and choose **`New, Class`** to create a Java client named `FindScoreApp` in package **`chapter41`**, as shown in Listing 41.2.



```
<?xml version='1.0' encoding='UTF-8'?><!-- Published by JAX-WS RI at http://jax-ws.de
<types>
<xsd:schema>
<xsd:import namespace="http://chapter45/" schemaLocation="http://localhost:8080/WebSe
</xsd:schema>
</types>
<message name="findScore">
<part name="parameters" element="tns:findScore" />
</message>
<message name="findScoreResponse">
<part name="parameters" element="tns:findScoreResponse" />
</message>
<portType name="ScoreService">
<operation name="findScore">
<input wsam:Action="http://chapter45/ScoreService/findScoreRequest" message="tns:find
<output wsam:Action="http://chapter45/ScoreService/findScoreResponse" message="tns:fi
</operation>
</portType>
```

FIGURE 41.8 The .wsdl file describes Web services to clients.

LISTING 41.2 FindScoreApp.java

```java
package chapter41;

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.GridPane;
import javafx.stage.Stage;
import myWebservice.ScoreWebService;
import myWebservice.ScoreService;

public class FindScoreApp extends Application {
  // Declare a service object and a proxy object
  private ScoreWebService scoreWebService = new ScoreWebService();
  private ScoreService proxy
    = scoreWebService.getScoreServicePort();

  private Button btGetScore = new Button("Get Score");
  private TextField tfName = new TextField();
  private TextField tfScore = new TextField();

  public void start(Stage primaryStage) {
    GridPane gridPane = new GridPane();
    gridPane.setHgap(5);
    gridPane.add(new Label("Name"), 0, 0);
    gridPane.add(new Label("Score"), 0, 1);
    gridPane.add(tfName, 1, 0);
    gridPane.add(tfScore, 1, 1);
    gridPane.add(btGetScore, 1, 2);

    // Create a scene and place the pane in the stage
    Scene scene = new Scene(gridPane, 250, 250);
    primaryStage.setTitle("FindScoreApp"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

38      btGetScore.setOnAction(e -> getScore());
39    }
40
41    private void getScore() {
42      try {
43        // Get student score
44        double score = proxy.findScore(tfName.getText().trim());
45
46        // Display the result
47        if  (score < 0)
48          tfScore.setText("Not found");
49        else
50          tfScore.setText(new Double(score).toString());
51        }
52        catch(Exception ex) {
53          ex.printStackTrace();
54      }
55    }
56  }
```

The program creates a Web service object (line 11) and creates a proxy object (line 12) to interact with the Web service.

To find a score for a student, the program invokes the remote method **findScore** on the proxy object (line 39).

41.5 Passing and Returning Arguments

The Simple Object Access Protocol (SOAP) can be used to send and return values to and from a Web service.

Key Point

In the preceding example, a Web service client you created invokes the **findScore** method with a string argument, and the Web service executes the method and returns a score as a **double** value. How does this work? It is the *Simple Object Access Protocol* (SOAP) that facilitates communications between the client and the server.

SOAP is based on XML. The message between the client and the server is described in XML. Figure 41.9 shows the SOAP request and SOAP response for the **findScore** method.

When invoking the **findScore** method, a *SOAP request* is sent to the server. The request contains the information about the method and the argument. As shown in Figure 41.9, the XML text

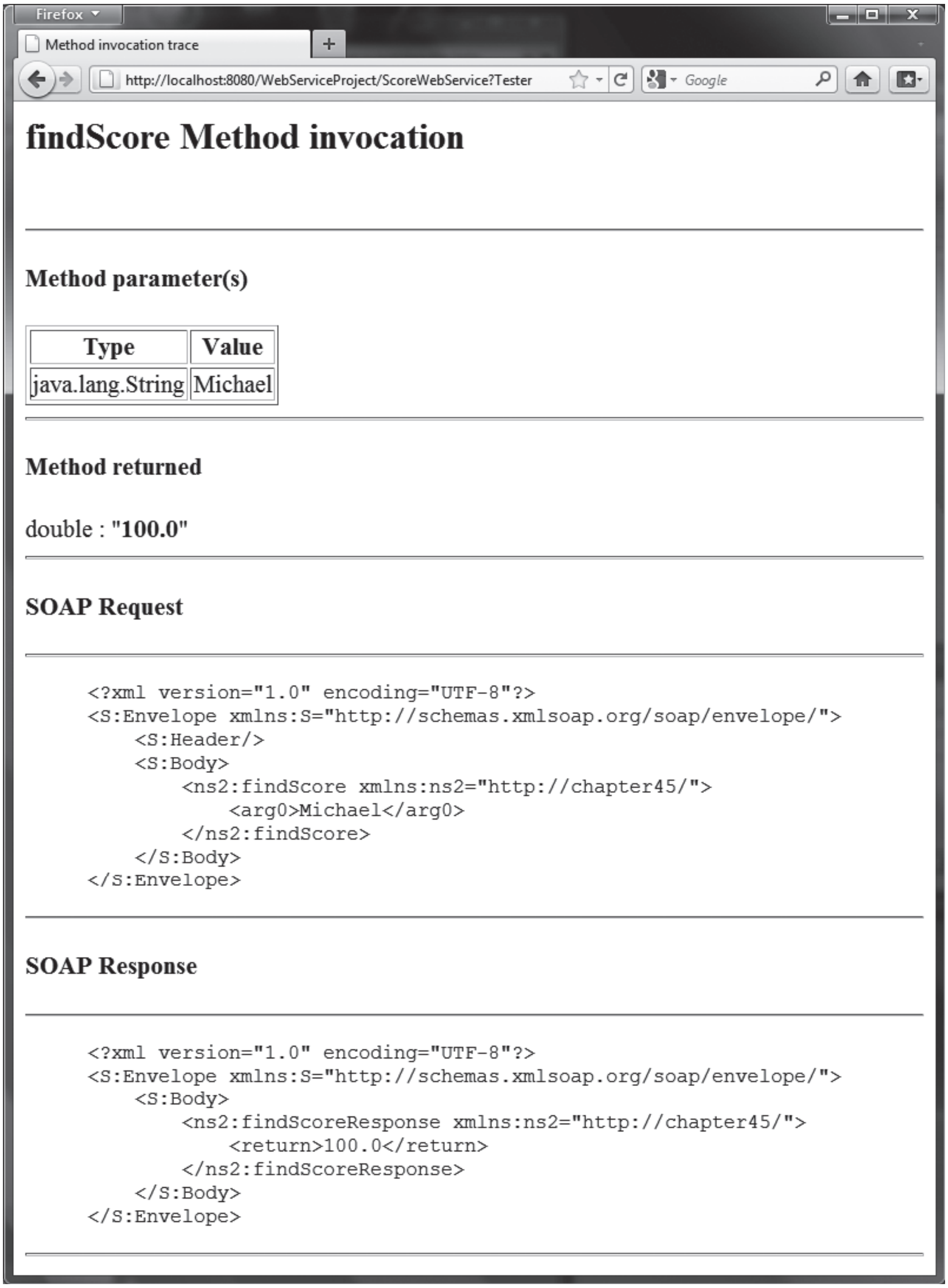
```
<ns1:findScore>
   <arg0>Michael</arg0>
</ns1:findScore>
```

specifies that the method **findScore** is called with argument **Michael**.

Upon receiving the SOAP request, the Web service parses it. After parsing it, the Web service invokes an appropriate method with specified arguments (if any) and sends the response back in a *SOAP response*. As shown in Figure 41.9, the XML text

```
<ns1:findScoreResponse>
   <return>100.0</return>
</ns1:findScoreResponse>
```

specifies that the method returns **100.0**.


Firefox
Method invocation trace
http://localhost:8080/WebServiceProject/ScoreWebService?Tester
Google

findScore Method invocation

Method parameter(s)

Type	Value
java.lang.String	Michael

Method returned

double : **"100.0"**

SOAP Request

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Header/>
    <S:Body>
        <ns2:findScore xmlns:ns2="http://chapter45/">
            <arg0>Michael</arg0>
        </ns2:findScore>
    </S:Body>
</S:Envelope>
```

SOAP Response

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:findScoreResponse xmlns:ns2="http://chapter45/">
            <return>100.0</return>
        </ns2:findScoreResponse>
    </S:Body>
</S:Envelope>
```

FIGURE 41.9 The client request and server response are described in XML.

The proxy object receives the SOAP response from the Web service and parses it. This process is illustrated in Figure 41.10.

Can you pass an argument of any type between a client and a Web service? No. SOAP supports only primitive types, wrapper types, arrays, `String`, `Date`, `Time`, `List`, and several other types. It also supports certain custom classes. An object that is sent to or from a server is serialized into XML. The process of serializing/deserializing objects, called

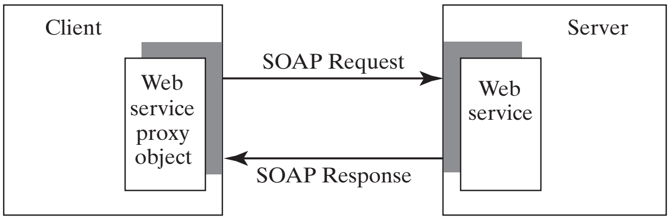


FIGURE 41.10 A proxy object sends SOAP requests and receives SOAP responses.

XML serialization/deserialization, is performed automatically. For a custom class to be used with Web methods, the class must meet the following requirements:

1. The class must have a no-arg constructor.
2. Instance variables that should be serialized must have public get and set methods. The classes of these variables must be supported by SOAP.

To demonstrate how to pass an object argument of a custom class, Listing 41.3 defines a Web service class named `AddressService` with two remote methods:

- `getAddress(String firstName, String lastName)` that returns an `Address` object for the specified `firstName` and `lastName`.
- `storeAddress(Address address)` that stores a `Student` object to the database.

Address information is stored in a table named `Address` in the database. The `Address` class was defined in Listing 42.12, Address.java. An `Address` object can be passed to or returned from a remote method, since the `Address` class has a no-arg constructor with get and set methods for all its properties.

Here are the steps to create a Web service named `AddressService` and the `Address` class in the project.

1. Right-click the `WebServiceProject` node in the project pane to display a context menu. Choose **`New, Web Service`** to display the New Web Service dialog box.
2. In the Web Service Name field, enter `AddressService`. In the Package field, enter `chapter41`. Click *Finish* to create the service class.
3. Right-click the `WebServiceProject` node in the project pane to display a context menu. Choose **`New, Java Class`** to display the New Java Class dialog box.
4. In the Class Name field, enter `Address`. In the Package field, enter `chapter37`. Click *Finish* to create the class.

The `Address` class is the same as shown in Listing 37.12. Complete the `AddressService` class as shown in Listing 41.3.

LISTING 41.3 AddressService.java

```
package chapter41;

import chapter37.Address;
import java.sql.*;
import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService(name = "AddressService",
  serviceName = "AddressWebService")
```

```
public class AddressService {
  // statement1 for retrieving an address and statement2 for storing
  private PreparedStatement statement1;

  // statement2 for storing an address
  private PreparedStatement statement2;

  public AddressService() {
    initializeJdbc();
  }

  @WebMethod(operationName = "getAddress")
  public Address getAddress(String firstName, String lastName) {
    try {
      statement1.setString(1, firstName);
      statement1.setString(2, lastName);
      ResultSet resultSet = statement1.executeQuery();

      if (resultSet.next()) {
        Address address = new Address();
        address.setFirstName(resultSet.getString("firstName"));
        address.setLastName(resultSet.getString("lastName"));
        address.setMi(resultSet.getString("mi"));
        address.setTelephone(resultSet.getString("telephone"));
        address.setFirstName(resultSet.getString("email"));
        address.setCity(resultSet.getString("telephone"));
        address.setState(resultSet.getString("state"));
        address.setZip(resultSet.getString("zip"));
        return address;
      }
      else
        return null;
    } catch (SQLException ex) {
      ex.printStackTrace();
    }

    return null;
  }

  @WebMethod(operationName = "storeAddress")
  public void storeAddress(Address address) {
    try {
      statement2.setString(1, address.getLastName());
      statement2.setString(2, address.getFirstName());
      statement2.setString(3, address.getMi());
      statement2.setString(4, address.getTelephone());
      statement2.setString(5, address.getEmail());
      statement2.setString(6, address.getStreet());
      statement2.setString(7, address.getCity());
      statement2.setString(8, address.getState());
      statement2.setString(9, address.getZip());
      statement2.executeUpdate();
    } catch (SQLException ex) {
    ex.printStackTrace();
    }
  }

  /** Initialize database connection */
  public void initializeJdbc() {
```

```
69      try {
70        Class.forName("com.mysql.jdbc.Driver");
71
72        // Connect to the sample database
73        Connection connection = DriverManager.getConnection(
74          "jdbc:mysql://localhost/javabook", "scott", "tiger");
75
76        statement1 = connection.prepareStatement(
77          "select * from Address where firstName = ? and lastName = ?");
78        statement2 = connection.prepareStatement(
79          "insert into Address "  +
80          "(lastName, firstName, mi, telephone, email, street, city, "
81          + "state, zip) values (?, ?, ?, ?, ?, ?, ?, ?, ?)");
82      } catch (Exception ex) {
83        ex.printStackTrace();
84      }
85    }
86  }
```

The new Web service is named `AddressWebService` (line 9) for the `AddressService` class.

When the service is deployed, the constructor (lines 17–19) of `AddressWebService` is invoked to initialize a database connection and create prepared `statement1` and `statement2` (lines 68–85).

The `findAddress` method searches the address in the `Address` table for the specified `firstName` and `lastName`. If found, the address information is returned in an `Address` object (lines 29–38). Otherwise, the method returns `null` (line 41).

The `storeAddress` method stores the address information from the `Address` object into the database (lines 52–61).

Note
Don't forget that you have to add the MySQL library to the `WebServiceProject` for this example to run.

Before you can use the service, deploy it. Right-click the `WebServiceProject` node in the Project to display a context menu and choose **`Deploy`**.

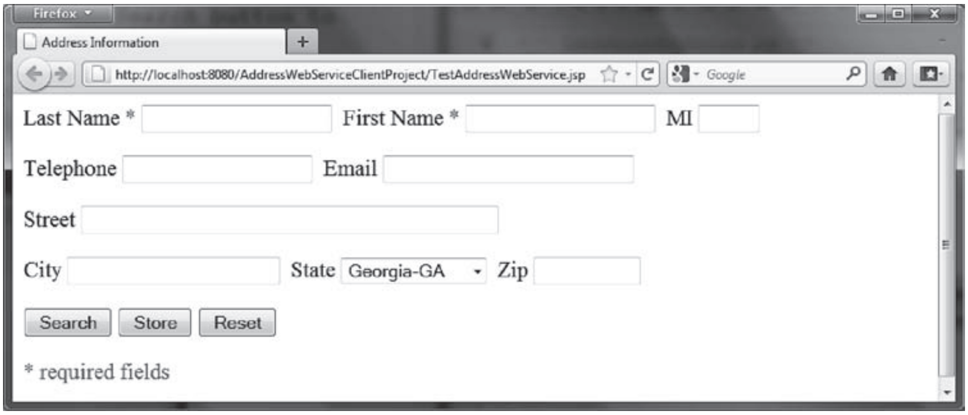
Now you are ready to develop a Web client that uses the `AddressWebService`. The client is a JSP program, as shown in Figure 41.11. The program has two functions. First, the user can enter the last name and first name and click the *Search* button to search for a record, as shown in Figure 41.12. Second, the user can enter the complete address information and click the *Store* button to store the information to the database, as shown in Figure 41.13.

Let us create a project for the client. The project named `AddressWebServiceClientProject` can be created as follows:

1. Choose **`File, New Project`** to display the New Web Application dialog box. In the New Web Application dialog box, choose **`Java Web`** in the Categories pane and choose **`Web Application`** in the Projects pane. Click *Next* to display the Name and Location dialog box.

2. Enter `AddressWebServiceClientProject` as the project name, specify the location where you want the project to be stored, and uncheck the *Set as Main Project* check box. Click *Next* to display the Server and Settings dialog box.

3. Choose **`GlassFish Server 4`** in the Server field, and **`Java EE 7 Web`** as in the Java EE Version field, and click *Finish* to create the project.


Firefox
Address Information
http://localhost:8080/AddressWebServiceClientProject/TestAddressWebService.jsp
Google
Search
Store
Reset

Last Name * First Name * MI

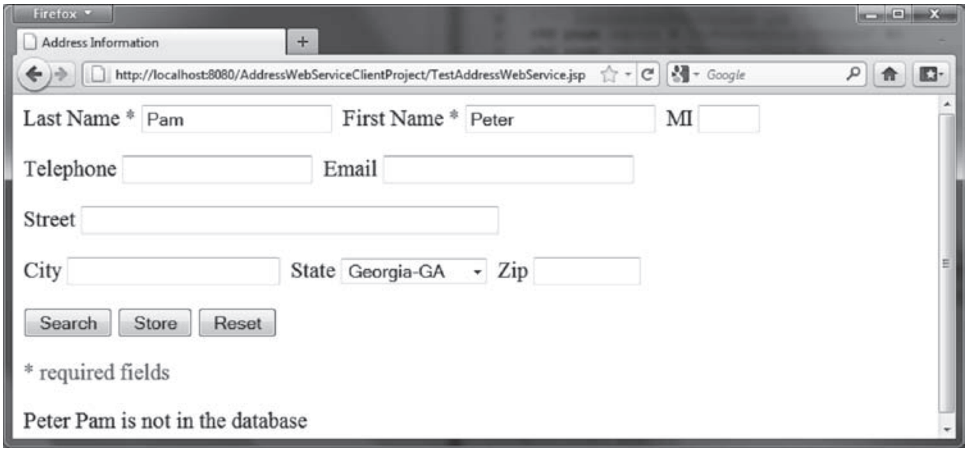
Telephone Email

Street

City State Georgia-GA Zip

* required fields

FIGURE 41.11 The `TestAddressWebService` page allows the user to search and store addresses.


Firefox
Address Information
http://localhost:8080/AddressWebServiceClientProject/TestAddressWebService.jsp
Google
Search
Store
Reset

Last Name * Pam First Name * Peter MI

Telephone Email

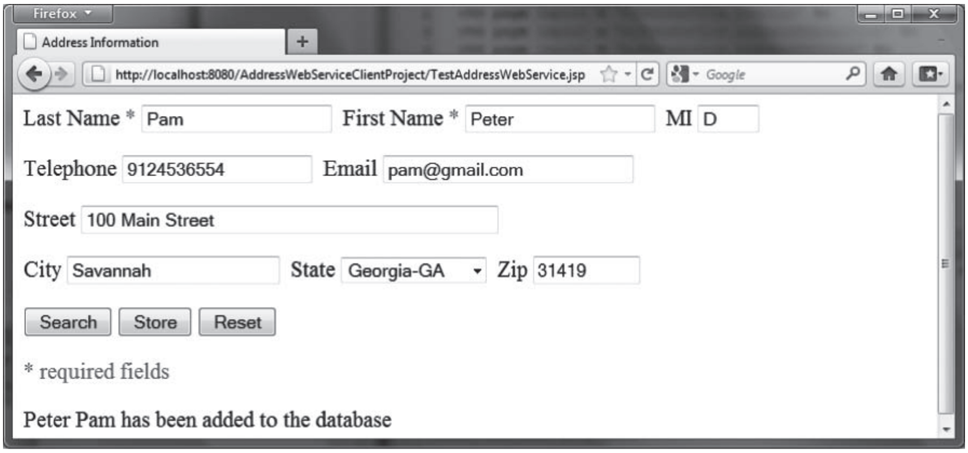
Street

City State Georgia-GA Zip

* required fields

Peter Pam is not in the database

FIGURE 41.12 The *Search* button finds and displays an address.


Firefox
Address Information
http://localhost:8080/AddressWebServiceClientProject/TestAddressWebService.jsp
Google
Search
Store
Reset

Last Name * Pam First Name * Peter MI D

Telephone 9124536554 Email pam@gmail.com

Street 100 Main Street

City Savannah State Georgia-GA Zip 31419

* required fields

Peter Pam has been added to the database

FIGURE 41.13 The *Store* button stores the address to the database.

You need to create a Web service reference to this project. The reference will enable you to create a proxy object to interact with the Web service. Here are the steps to create a Web service reference:

1. Right-click the `AddressWebServiceClientProject` node in the Project pane to display a context menu. Choose **`New, Web Service Client`** to display the New Web Service Client dialog box.

2. Check the *WSDL URL radio* button and enter http://localhost:8080/WebServiceProject/AddressWebService?WSDL in the WSDL URL field.

3. Enter `myWebservice` in the package name field and choose **`JAX-WS`** as the JAX version. Click *Finish* to generate the Web service reference.

Now a reference to `AddressWebService` is created. Note this process also copies `Address.java` to the client project, as shown in Figure 41.14.

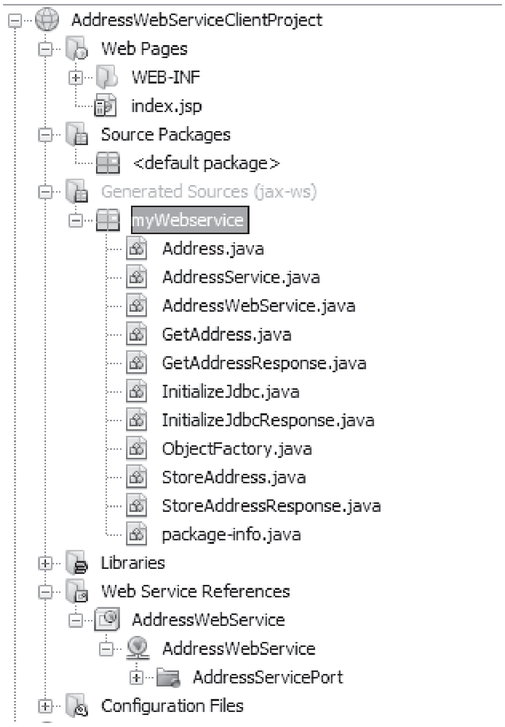


FIGURE 41.14 Address.java is automatically copied to the Web service client reference package.

Create a JSP named `TestAddressWebService` in the `AddressWebServiceClientProject` project, as shown in Listing 41.4.

LISTING 41.4 TestAddressWebService.jsp

```
<!-- TestAddressWebService.jsp -->
<%@ page import = "myWebservice.Address" %>
<%@ page import = "myWebservice.AddressWebService" %>
<%@ page import = "myWebservice.AddressService" %>
<jsp:useBean id = "addressId"
  class = "myWebservice.Address" scope = "session"></jsp:useBean>
<jsp:setProperty name = "addressId" property = "*" />

```

```
<html>
<head>
  <title>Address Information</title>
</head>
<body>
  <form method = "post" action = "TestAddressWebService.jsp">
  Last Name <font color = "#FF0000">*</font>
  <input type = "text" name = "lastName"
    <%if (addressId.getLastName() != null) {
      out.print("value = \"" + addressId.getLastName() + "\"");}%>
    size = "20" />&nbsp;

  First Name <font color = "#FF0000">*</font>
  <input type = "text" name = "firstName"
    <%if (addressId.getFirstName() != null) {
      out.print("value = \"" + addressId.getFirstName() + "\"");}%>
    size = "20" />&nbsp;

  MI
  <input type = "text" name = "mi"
    <%if (addressId.getMi() != null) {
      out.print("value = \"" + addressId.getMi() + "\" "); } %>
    size = "3" />&nbsp;

  <p>Telephone
  <input type = "text" name = "telephone"
    <%if (addressId.getTelephone() != null) {
      out.print("value = \"" + addressId.getTelephone() + "\" ");}%>
    size = "20" />&nbsp;

   Email
   <input type = "text" name = "email"
     <%if (addressId.getEmail() != null) {
       out.print("value = \"" + addressId.getEmail() + "\" ");}%>
     size = "28" />&nbsp;
   </p>

   <p>Street
   <input type = "text" name = "street"
     <%if (addressId.getStreet() != null) {
       out.print("value = \"" + addressId.getStreet() + "\" ");}%>
     size = "50" />&nbsp;
   </p>

   <p>City
   <input type = "text" name = "city"
     <%if (addressId.getCity() != null) {
       out.print("value = \"" + addressId.getCity() + "\" ");}%>
   size = "23" />&nbsp;

  State
  <select size = "1" name = "state">
    <option value = "GA">Georgia-GA</option>
    <option value = "OK">Oklahoma-OK</option>
    <option value = "IN">Indiana-IN</option>
  </select>&nbsp;

  Zip
  <input type = "text" name = "zip"
    <%if (addressId.getZip() != null) {
      out.print("value = \"" + addressId.getZip() + "\" "); } %>
```

```
70        size = "9" />&nbsp;
71      </p>
72
73      <p><input type = "submit" name = "Submit" value = "Search">
74        <input type = "submit" name = "Submit" value = "Store">
75        <input type = "reset" value = "Reset">
76      </p>
77      </form>
78      <p><font color = "#FF0000">* required fields</font></p>
79
80      <%
81      if (request.getParameter("Submit") != null) {
82        AddressWebService addressWebService = new AddressWebService();
83        AddressService proxy = addressWebService.getAddressServicePort();
84
85        if (request.getParameter("Submit").equals("Store")) {
86          proxy.storeAddress(addressId);
87          out.println(addressId.getFirstName() + " " +
88            addressId.getLastName() + " has been added to the database");
89        }
90        else if (request.getParameter("Submit").equals("Search")) {
91          Address address = proxy.getAddress(addressId.getFirstName(),
92            addressId.getLastName());
93          if (address == null)
94            out.print(addressId.getFirstName() + " " +
95              addressId.getLastName() + " is not in the database");
96          else
97            addressId = address;
98        }
99      }
100     %>
101   </body>
102   </html>
```

Lines 2–4 import the classes for the JSP page. The `Address` class (line 2) was created in the `WebServiceProject` and was automatically copied to the `AddressWebServiceClientProject` project when a Web service reference for `AddressWebService` was created. A JavaBeans object for `Address` was created and associated with input parameters in lines 5–7.

The UI interface was laid in the form (lines 14–77). The action for the two buttons *Search* and *Store* invokes the same page TestAddressWebService.jsp (line 14).

When a button is clicked, a proxy object for `AddressWebService` is obtained (lines 82–83). For the *Store* button, the proxy object invokes the `storeAddress` method to add an address to the database (line 86). For the *Search* button, the proxy object invokes the `getAddress` method to return an address (lines 91–92). If no address is found for the specified first and last names, the returned address is `null` (line 93).

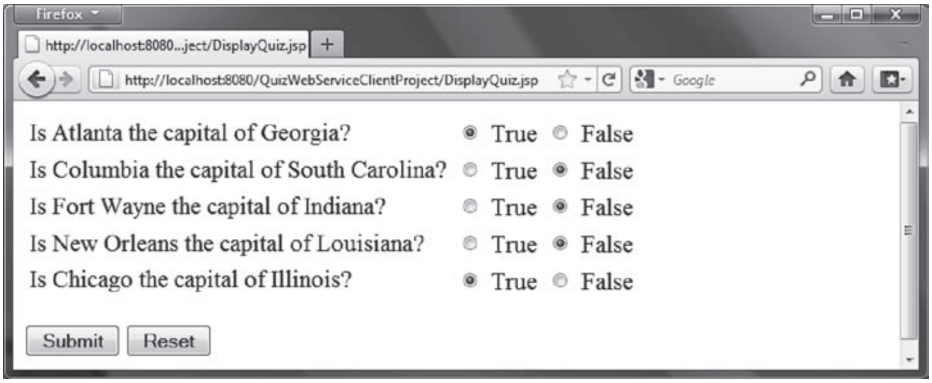
41.6 Web Service Session Tracking

You can use the `HttpSession` *interface to session tracking for Web.*

Key Point

Section 37.8.3, Session Tracking Using the Servlet API, introduced session tracking for servlets using the `javax.servlet.http.HttpSession` interface. You can use `HttpSession` to implement session tracking for Web services. To demonstrate this, consider an example that generates random True/False questions for the client and grades the answers on these questions for the client.

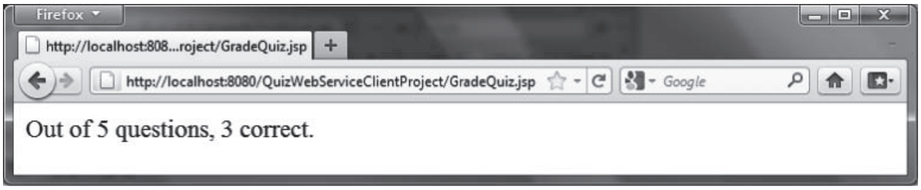
The Web client consists of two JSP pages: DisplayQuiz.jsp and GradeQuiz.jsp. The `DisplayQuiz` page invokes the service method `getQuestion()` to display the questions, as shown in Figure 41.15. When you click the *Submit* button, the program invokes the service



http://localhost:8080/QuizWebServiceClientProject/DisplayQuiz.jsp

Is Atlanta the capital of Georgia? ● True ○ False
Is Columbia the capital of South Carolina? ○ True ● False
Is Fort Wayne the capital of Indiana? ○ True ● False
Is New Orleans the capital of Louisiana? ○ True ● False
Is Chicago the capital of Illinois? ● True ○ False

FIGURE 41.15 The *Submit* button submits the answers for grading.



http://localhost:8080/QuizWebServiceClientProject/GradeQuiz.jsp

Out of 5 questions, 3 correct.

FIGURE 41.16 The answers are graded and displayed.

method `gradeQuiz` to grade the answers. The result is displayed in the `GradeQuiz` page, as shown in Figure 41.16.

Why is session tracking needed for this project? Each time a client displays a quiz, it creates a randomly reorder the quiz for the client. Each client gets a different quiz every time the `DisplayQuiz` page is refreshed. When the client submits the answer, the Web service checks the answer against the previously generated quiz. So the quiz has to be stored in the session.

For convenience, let us create the Web service class named `QuizService` in the `WebServiceProject` in package `chapter41`. Listing 41.5 gives the program.

LISTING 41.5 QuizService.java

```java
package chapter41;

import javax.jws.WebMethod;
import javax.jws.WebService;
import java.util.List;
import java.util.ArrayList;
import com.sun.xml.ws.developer.servlet.HttpSessionScope;

@HttpSessionScope
@WebService(name = "QuizService", serviceName = "QuizWebService")
public class QuizService {
  private ArrayList<Object[]> quiz = new ArrayList<Object[]>();

  public QuizService() {
    // Initialize questions and answers
    quiz.add(new Object[]{
      "Is Atlanta the capital of Georgia?", true});
    quiz.add(new Object[]{
      "Is Columbia the capital of South Carolina?", true});
    quiz.add(new Object[]{
      "Is Fort Wayne the capital of Indiana?", false});
22        quiz.add(new Object[]{
23          "Is New Orleans the capital of Louisiana?", false});
24        quiz.add(new Object[]{
25          "Is Chicago the capital of Illinois?", false});
26
27        // Shuffle to generate a random quiz for a client
28        java.util.Collections.shuffle(quiz);
29      }
30
31      @WebMethod(operationName = "getQuestions")
32      public java.util.List<String> getQuestions() {
33
34        // Extract questions from quiz
35        List<String> questions = new ArrayList<String>();
36        for  (int i = 0; i < quiz.size(); i++) {
37          questions.add((String)(quiz.get(i)[0]));
38        }
39
40        return questions; // Return questions in the quiz
41      }
42
43      @WebMethod(operationName = "gradeQuiz")
44      public List<Boolean> gradeQuiz(List<Boolean> answers) {
45        List<Boolean> result = new ArrayList<Boolean>();
46        for (int i = 0; i < quiz.size(); i++)
47          result.add(quiz.get(i)[1] == answers.get(i));
48
49        return result;
50      }
51    }
```

The Web service class named `QuizService` contains two methods `getQuestions` and `gradeQuiz`. The new Web service is named `QuizWebService` (line 10).

The annotation `@HttpSessionScope` (line 9) is new in JAX-WS 2.2, which enables the Web service automatically maintains a separate instance for each client session. To use this annotation, you have add JAX-WS 2.2 into your project's library. This can be done by clicking the **Library** node in the project and select **Add Library**.

Assume five True/False questions are available from the service. The quiz is stored in an `ArrayList` (lines 16–25).

Each element in the list is an array with two values. The first value is a string that describes the question and the second is a Boolean value indicating whether the answer should be true or false.

A new quiz is generated in the constructor and the quiz is shuffled using the shuffle method in the Collections class (line 28).

The `getQuestions` method (lines 31–40) returns questions in a list. The questions are extracted from the quiz (lines 34–37) and are returned (line 39).

The `gradeQuiz` method (lines 42–49) checks the `answers` from the client with the answers in the quiz. The client's answers are compared with the key, and the result of the grading is stored in a list. Each element in the list is a Boolean value that indicates whether the answer is correct or incorrect (lines 44–46).

After creating and publishing the Web service, let us create a project for the client. The project named `QuizWebServiceClientProject` can be created as follows:

1. Choose **File, New Project** to display the New Web Application dialog box.

2. In the New Web Application dialog box, choose **Java Web** in the Categories pane and choose **Web Application** in the Projects pane. Click *Next* to display the Name and Location dialog box.

3. Enter QuizWebServiceClientProject as the project name, specify the location where you want the project to be stored, and uncheck the *Set as Main Project* check box. Click *Next* to display the Server and Settings dialog box.
4. Choose **GlassFish Server 4** in the Server field, and **Java EE 7 Web** as in the Java EE Version field, and click Finish to create the project.

To use QuizWebService, you need to create a Web service client as follows:

1. Right-click the QuizWebServiceClientProject project in the Project pane to display a context menu. Choose **New, Web Service Client** to display the New Web Service Client dialog box.
2. Check the *WSDL URL radio* button and enter http://localhost:8080/WebServiceProject/QuizWebService?WSDL in the WSDL URL field.
3. Enter myWebservice in the Package field. Click *Finish* to create the reference for QuizWebService.

Now a reference to QuizWebService is created. You can create a proxy object to access the remote methods in QuizService. Listings 41.6 and 41.7 show DisplayQuiz.jsp and GradeQuiz.jsp.

Listing 41.6 DisplayQuiz.jsp

```
<!-- DisplayQuiz.jsp -->
<%@ page import = "myWebservice.QuizWebService" %>
<%@ page import = "myWebservice.QuizService" %>
<jsp:useBean id = "quizWebService" scope = "session"
  class = "myWebservice.QuizWebService">
</jsp:useBean>

<html>
<body>
  <%
  QuizService proxy = quizWebService.getQuizServicePort();
  java.util.List<String> questions =
    (java.util.ArrayList<String>)(proxy.getQuestions());
  %>
  <form method = "post" action = "GradeQuiz.jsp">
  <table>
    <% for (int i = 0; i < questions.size(); i++) {%>
    <tr>
    <td>
      <label><%= questions.get(i) %></label>
    </td>
    <td>
      <input type = "radio" name = <%= "question" + i%>
        value = "True" /> True
    </td>
    <td>
      <input type = "radio" name = <%= "question" + i%>
      value = "False" /> False
    </td>
    </tr>
    <%}%>
  </table>
  <p><input type = "submit" name = "Submit" value = "Submit">
    <input type = "reset" value = "Reset">
```

```
35     </p>
36     </form>
37   </body>
38   </html>
```

This page generates a quiz by invoking the `getQuestions()` in lines 12–13. The questions are displayed in a table with radio buttons (lines 16–32). Clicking the *Submit* button invokes GradeQuiz.jsp.

Listing 41.7 GradeQuiz.jsp

```
 1   <!-- GradeQuiz.jsp -->
 2   <%@ page import = "myWebservice.QuizWebService" %>
 3   <%@ page import = "myWebservice.QuizService" %>
 4   <jsp:useBean id = "quizWebService" scope = "session"
 5     class = "myWebservice.QuizWebService">
 6   </jsp:useBean>
 7
 8   <html>
 9   <body>
10   <%
11   QuizService proxy = quizWebService.getQuizServicePort();
12   java.util.List<String> quiz = proxy.getQuestions();
13
14   // Get the answer from the DisplayQuiz page
15   java.util.List<Boolean> answers = new java.util.ArrayList<Boolean>();
16   for (int i = 0; i < quiz.size(); i++) {
17     String trueOrFalse = request.getParameter("question" + i);
18     if (trueOrFalse.equals("True"))
19       answers.add(true); // Answered true
20     else if (trueOrFalse.equals("False"))
21       answers.add(false); // Answered false
22   }
23
24   // Grade answers
25   java.util.List<Boolean> result = proxy.gradeQuiz(answers);
26
27   // Find the correct count
28   int correctCount = 0;
29   for (int i = 0; i < result.size(); i++) {
30     if (result.get(i))
31       correctCount++;
32   }
33   %>
34
35   Out of <%= result.size() %> questions, <%= correctCount %> correct.
36   </body>
37   </html>
```

This page collects the answers passed from the HTML form from the `DisplayQuiz` page (lines 15–21), invokes the `gradeQuiz` method to grade the quiz (line 25), finds the correct count (lines 28–31), and displays the result (line 35).

Note

You need to answer all five questions before clicking the *Submit* button. A runtime error will occur if a radio button is not checked. You can fix this problem in Exercise 41.5.

Check Point

41.6.1 What is the annotation to specify a Web service? What is the annotation to specify a Web method?

41.6.2 How do you deploy a Web service in NetBeans?

41.6.3 Can you test a Web service from a client?

41.6.4 How do you create a Web service reference for a client?

41.6.5 What is WSDL? What is SOAP? What is a SOAP request? What is a SOAP response?

41.6.6 Can you pass primitive type arguments to a remote method? Can you pass any object type to a remote method? Can you pass an argument of a custom type to a remote method?

41.6.7 How do you obtain an `HttpSession` object for tracking a Web session?

41.6.8 Can you create two Web service references in one package in the same project in NetBeans?

41.6.9 What happens if you don't clone the quiz in lines 40–41 in Listing 41.5, QuizService.java?

Key Terms

`@WebService`	41-3	publishing a Web service	41-2
`@WebMethod`	41-3	Web service	41-2
consuming a Web service	41-2	Web service client reference	41-15
proxy object	41-2	WSDL	41-6

Chapter Summary

1. Web services enable a Java program on one system to invoke a method in an object on another system.

2. Web services are platform and language independent. You can develop and use Web services using any language.

3. Web services run on the Web using HTTP. SOAP is a popular protocol for implementing Web services.

4. The server needs to make the service available to the client, known as *publishing a Web service*. Using a Web service from a client is known as *consuming a Web service*.

5. A client interacts with a Web service through a *proxy object*. The proxy object facilitates the communication between the client and the Web service.

6. You need to use Java annotation `@WebService` to annotate a Web service and use annotation `@WebMethod` to annotate a remote method.

7. A Web service class may have an unlimited number of remote methods.

8. After a Web service is published, you can write a client program to use it. You have to first create a Web client reference. From the reference, you create a proxy object for facilitating communication between a server and a client.

9. WSDL stands for *Web Service Description Language*. A .wsdl file is an XML file that describes the available Web service to the client—i.e., the remote methods, their parameters and return value types, and so on.

10. The message between the client and the server is described in XML. A SOAP request describes the information that is sent to the Web service and a SOAP response describes the information that is received from the Web service.

11. The objects passed between client and Web service are serialized in XML. Not all object types are supported by SOAP.

12. You can track sessions in Web services using the `HttpSession` in the same way as in servlets.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

Programming Exercises

MyProgrammingLab™

***41.1** (*Get a score from a database table)* Suppose the scores are stored in the `Scores` table. The table was created as follows:

```
create table Scores (name varchar(20),
  score number, permission boolean);

insert into Scores values (‘John’, 90.5, 1);
insert into Scores values (’Michael’, 100, 1);
insert into Scores values (’Michelle’, 100, 0);
```

Revise the `findScore` method in Listing 41.1, ScoreService.java, to obtain a score for the specified name. Note your program does not need the `permission` column; ignore it. The next exercise will need the `permission` column.

***41.2** (*Permission to find scores)* Revise the preceding exercise so that the `findScore` method returns `–1` if permission is `false`. Add an another method named `getPermission(String name)` that returns `1`, `0`, or `–1`. The method returns `1` if the student is in the `Scores` table and permission is `true`, `0` if the student is in the `Scores` table and permission is `false`, and `–1` if the student is not in the `Scores` table.

***41.3** (*Compute loan*) You can compute a loan payment for a loan with the specified amount, the number of years, and the annual interest rate. Write a Web service with two remote methods for computing monthly payment and total payment. Write a client program that prompts the user to enter the loan amount, the number of years, and the annual interest rate.

***41.4** (*Web service visit count*) Write a Web service with a method named `getCount()` that returns the number of the times this method has been invoked from a client. Use a session to store the `count` variable.

***41.5** (*Quiz*) The user needs to answer all five questions before clicking the *Submit* button in the Quiz application in Section 41.6, Web Service Session Tracking. A runtime error will occur if a radio button is not checked. Fix this problem.

CHAPTER

42

2–4 Trees and B-Trees

Objectives

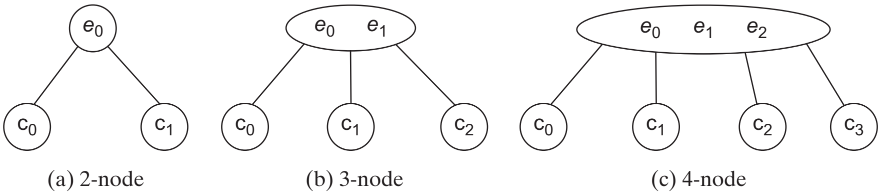
- To know what a 2–4 tree is (§42.1).
- To design the `Tree24` class that implements the `Tree` interface (§42.2).
- To search an element in a 2–4 tree (§42.3).
- To insert an element in a 2–4 tree and know how to split a node (§42.4).
- To delete an element from a 2–4 tree and know how to perform transfer and fusion operations (§42.5).
- To traverse elements in a 2–4 tree (§42.6).
- To implement and test the `Tree24` class (§§42.7–42.8).
- To analyze the complexity of the 2–4 tree (§42.9).
- To use B-trees for indexing large amount of data (§42.10).

42.1 Introduction

Key Point

A 2–4 tree, also known as a 2–3–4 tree, is a completely balanced search tree with all leaf nodes appearing on the same level.

In a 2–4 tree, a node may have one, two, or three elements as shown in Figure 42.1. An interior *2-node* contains one element and two children. An interior *3-node* contains two elements and three children. An interior *4-node* contains three elements and four children.



(a) 2-node (b) 3-node (c) 4-node

FIGURE 42.1 An interior node of a 2–4 tree has two, three, or four children.

Each child is a sub 2–4 tree, possibly empty. The root node has no parent, and leaf nodes have no children. The elements in the tree are distinct. The elements in a node are ordered such that

$$E(c_0) < e_0 < E(c_1) < e_1 < E(c_2) < e_2 < E(c_3)$$

where $E(c_k)$ denote the elements in c_k. Figure 42.2 shows an example of a 2–4 tree. c_k is called the *left subtree* of e_k and c_{k+1} is called the *right subtree* of e_k.

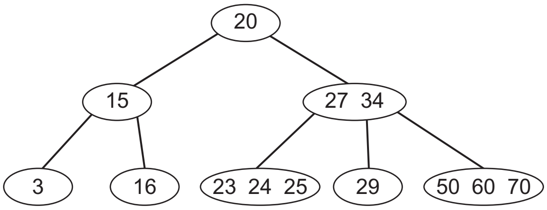


FIGURE 42.2 A 2–4 tree is a full complete search tree.

In a binary tree, each node contains one element. A 2–4 tree tends to be shorter than a corresponding binary search tree, since a 2–4 tree node may contain two or three elements.

Pedagogical Note

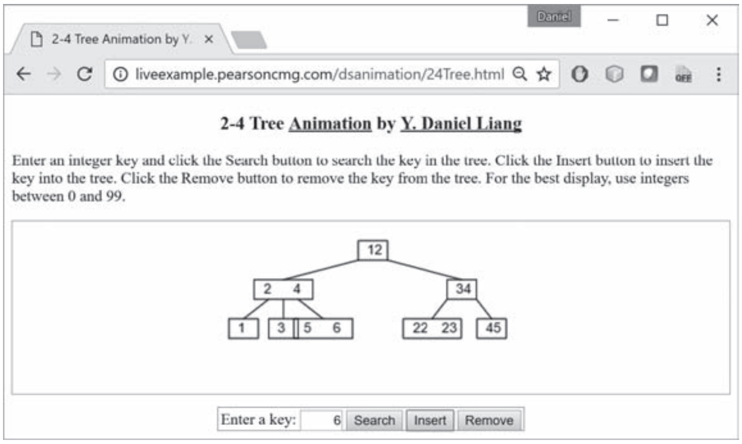
Run from http://liveexample.pearsoncmg.com/dsanimation/24Tree.html to see how a 2–4 tree works, as shown in Figure 42.3.

42.2 Designing Classes for 2–4 Trees

Key Point

The `Tree24` class defines a 2–4 tree and provides methods for searching, inserting, and deleting elements.

The `Tree24` class can be designed by implementing the `Tree` interface, as shown in Figure 42.4. The `Tree` interface was defined in Listing 27.3, Tree.java. The `Tree24Node` class defines tree nodes. The elements in the node are stored in a list named `elements` and the links to the child nodes are stored in a list named `child`, as shown in Figure 42.5.

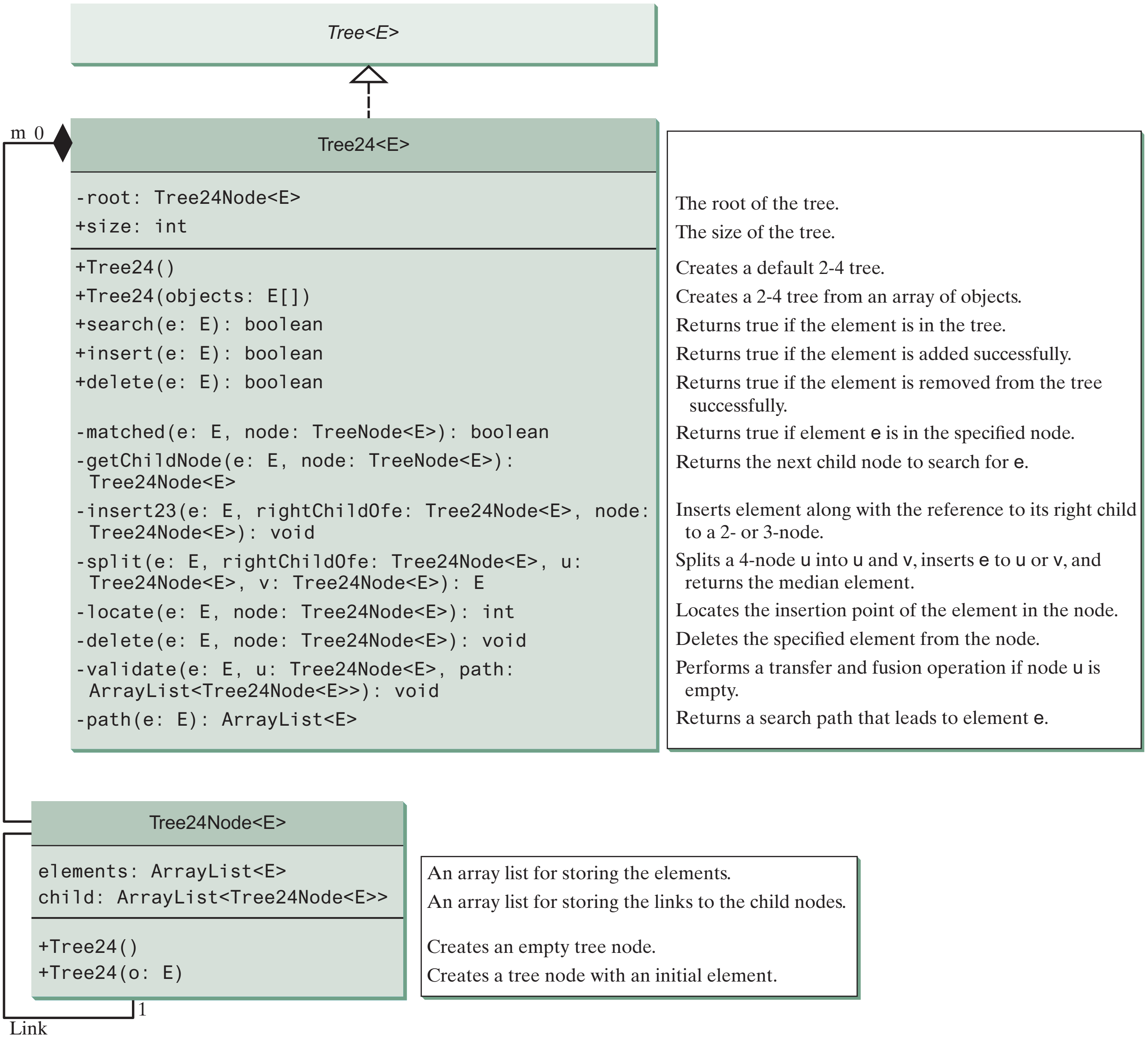


2-4 Tree Animation by Y. Daniel Liang

Enter an integer key and click the Search button to search the key in the tree. Click the Insert button to insert the key into the tree. Click the Remove button to remove the key from the tree. For the best display, use integers between 0 and 99.

Enter a key: 6 Search Insert Remove

FIGURE 42.3 The animation tool enables you to insert, delete, and search elements in a 2–4 tree visually.



Tree24<E>

`-root: Tree24Node<E>`	The root of the tree.
`+size: int`	The size of the tree.
`+Tree24()`	Creates a default 2-4 tree.
`+Tree24(objects: E[])`	Creates a 2-4 tree from an array of objects.
`+search(e: E): boolean`	Returns true if the element is in the tree.
`+insert(e: E): boolean`	Returns true if the element is added successfully.
`+delete(e: E): boolean`	Returns true if the element is removed from the tree successfully.
`-matched(e: E, node: TreeNode<E>): boolean`	Returns true if element e is in the specified node.
`-getChildNode(e: E, node: TreeNode<E>): Tree24Node<E>`	Returns the next child node to search for e.
`-insert23(e: E, rightChildOfe: Tree24Node<E>, node: Tree24Node<E>): void`	Inserts element along with the reference to its right child to a 2- or 3-node.
`-split(e: E, rightChildOfe: Tree24Node<E>, u: Tree24Node<E>, v: Tree24Node<E>): E`	Splits a 4-node u into u and v, inserts e to u or v, and returns the median element.
`-locate(e: E, node: Tree24Node<E>): int`	Locates the insertion point of the element in the node.
`-delete(e: E, node: Tree24Node<E>): void`	Deletes the specified element from the node.
`-validate(e: E, u: Tree24Node<E>, path: ArrayList<Tree24Node<E>>): void`	Performs a transfer and fusion operation if node u is empty.
`-path(e: E): ArrayList<E>`	Returns a search path that leads to element e.

Tree24Node<E>

`elements: ArrayList<E>`	An array list for storing the elements.
`child: ArrayList<Tree24Node<E>>`	An array list for storing the links to the child nodes.
`+Tree24()`	Creates an empty tree node.
`+Tree24(o: E)`	Creates a tree node with an initial element.

FIGURE 42.4 The `Tree24` class implements `Tree`.

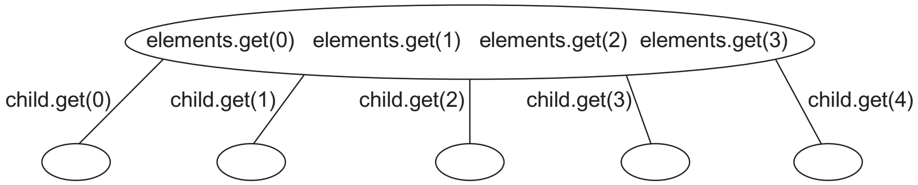


FIGURE 42.5 A 2–4 tree node stores the elements and the links to the child nodes in array lists.

Check Point

42.2.1 What is a 2–4 tree? What are a 2-node, 3-node, and 4-node?

42.2.2 Describe the data fields in the `Tree24` class and those in the `Tree24Node` class.

42.2.3 What is the minimum number of elements in a 2–4 tree of height 5? What is the maximum number of elements in a 2–4 tree of height 5?

42.3 Searching an Element

Key Point

Searching an element in a 2–4 tree is similar to searching an element in a binary tree. The difference is that you have to search an element within a node in addition to searching elements along the path.

To search an element in a 2–4 tree, you start from the root and scan down. If an element is not in the node, move to an appropriate subtree. Repeat the process until a match is found or you arrive at an empty subtree. The algorithm is described in Listing 42.1.

LISTING 42.1 Searching an Element in a 2–4 tree

```
 1  boolean search(E e) {
 2    current = root; // Start from the root
 3
 4    while (current != null) {
 5      if (match(e, current)) { // Element is in the node
 6        return true; // Element is found
 7      }
 8      else {
 9        current = getChildNode(e, current); // Search in a subtree
10      }
11    }
12    return false; // Element is not in the tree
13  }
```

The `match(e, current)` method checks whether element `e` is in the current node. The `getChildNode(e, current)` method returns the root of the subtree for further search. Initially, let `current` point to the root (line 2). Repeat searching for the element in the current node until `current` is `null` (line 4) or the element matches an element in the current node.

42.4 Inserting an Element into a 2–4 tree

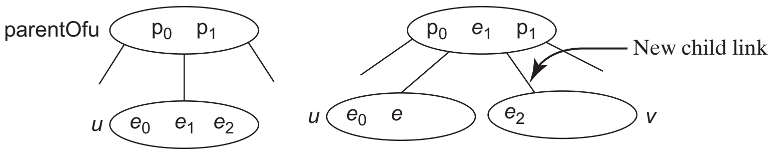
Key Point

Inserting an element involves locating a leaf node and inserting the element into the leaf node.

To insert an element `e` to a 2–4 tree, locate a leaf node in which the element will be inserted. If the leaf node is a 2-node or 3-node, simply insert the element into the node. If the node is a 4-node, inserting a new element would cause an *overflow*. To resolve overflow, perform a *split* operation as follows:

- Let `u` be the *leaf* 4-node in which the element will be inserted and `parentOfu` be the parent of `u`, as shown in Figure 42.6(a).

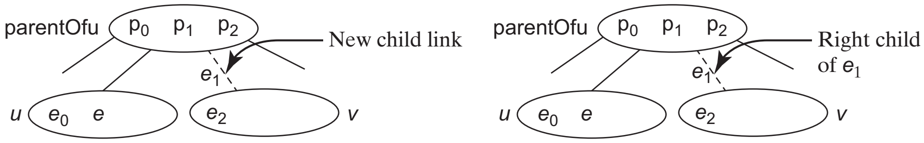
- Create a new node named *v*; move e_2 to *v*.
- If $e < e_1$, insert *e* to *u*; otherwise insert *e* to *v*. Assume $e_0 < e < e_1$, *e* is inserted into *u*, as shown in Figure 42.6(b).
- Insert e_1 along with its right child (i.e., *v*) to the parent node, as shown in Figure 42.6(b).



(a) Before inserting *e* (b) After inserting *e*

FIGURE 42.6 The splitting operation creates a new node and inserts the median element to its parent.

The parent node is a 3-node in Figure 42.6. So, there is room to insert *e* to the parent node. What happens if it is a 4-node, as shown in Figure 42.7? This requires that the parent node be split. The process is the same as splitting a leaf 4-node, except that you must also insert the element along with its right child.



(a) The parent is a 4-node (b) Inserting e_1 into the parent

FIGURE 42.7 Insertion process continues if the parent node is a 4-node.

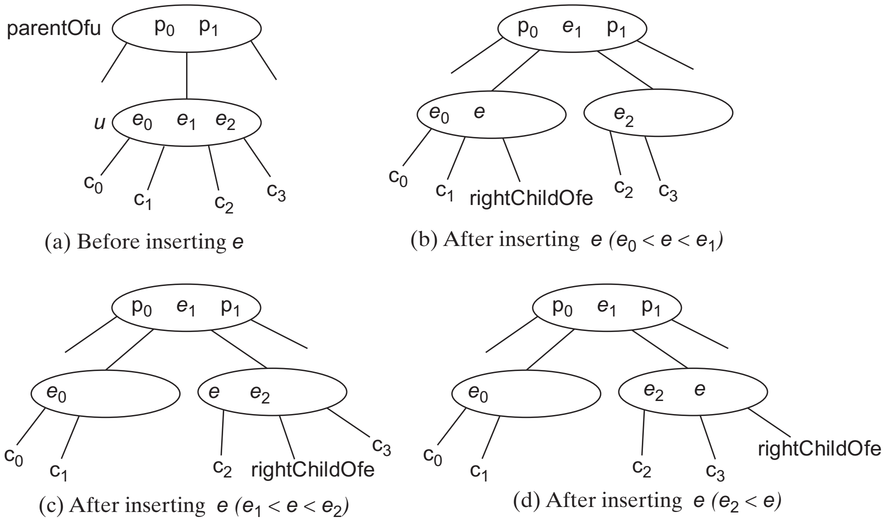
The algorithm can be modified as follows:

- Let *u* be the 4-node (*leaf or nonleaf*) in which the element will be inserted and *parentOfu* be the parent of *u*, as shown in Figure 42.8(a).
- Create a new node named *v*, move e_2 and its children c_2 and c_3 to *v*.
- If $e < e_1$, insert *e* along with its right child link to *u*; otherwise insert *e* along with its right child link to *v*, as shown in Figure 42.6(b), (c), (d) for the cases $e_0 < e < e_1$, $e_1 < e < e_2$, and $e_2 < e$, respectively.
- Insert e_1 along with its right child (i.e., *v*) to the parent node, recursively.

Listing 42.2 gives an algorithm for inserting an element.

LISTING 42.2 Inserting an Element to a 2–4 tree

```
public boolean insert(E e) {
  if (root == null)
    root = new Tree24Node<E>(e); // Create a new root for element
  else {
    Locate leafNode for inserting e
    insert(e, null, leafNode); // The right child of e is null
  }

  size++; // Increase size
  return true; // Element inserted
}

```



(a) Before inserting e

(b) After inserting e ($e_0 < e < e_1$)

(c) After inserting e ($e_1 < e < e_2$)

(d) After inserting e ($e_2 < e$)

FIGURE 42.8 An interior node may be split to resolve overflow.

```
13  private void insert(E e, Tree24Node<E> rightChildOfe,
14      Tree24Node<E> u) {
15    if (u is a 2- or 3- node) { // u is a 2- or 3-node
16      insert23(e, rightChildOfe, u); // Insert e to node u
17    }
18    else { // Split a 4-node u
19      Tree24Node<E> v = new Tree24Node<E>(); // Create a new node
20      E median = split(e, rightChildOfe, u, v); // Split u
21
22      if (u == root) { // u is the root
23        root = new Tree24Node<E>(median); // New root
24        root.child.add(u); // u is the left child of median
25        root.child.add(v); // v is the right child of median
26      }
27      else {
28        Get the parent of u, parentOfu;
29        insert(median, v, parentOfu); // Inserting median to parent
30      }
31    }
32  }
```

The `insert(E e, Tree24Node<E> rightChildOfe, Tree24Node<E> u)` method inserts element `e` along with its right child to node `u`. When inserting `e` to a leaf node, the right child of `e` is `null` (line 6). If the node is a 2- or 3-node, simply insert the element to the node (lines 15–17). If the node is a 4-node, invoke the `split` method to split the node (line 20). The `split` method returns the median element. Recursively invoke the `insert` method to insert the median element to the parent node (line 29). Figure 42.9 shows the steps of inserting elements **34**, **3**, **50**, **20**, **15**, **16**, **25**, **27**, **29**, and **24** into a 2–4 tree.

42.5 Deleting an Element from a 2–4 tree

Key Point

Deleting an element involves locating the node that contains the element and removing the element from the node.

To delete an element from a 2–4 tree, first search the element in the tree to locate the node that contains it. If the element is not in the tree, the method returns false. Let `u` be the node that contains the element and `parentOfu` be the parent of `u`. Consider three cases:

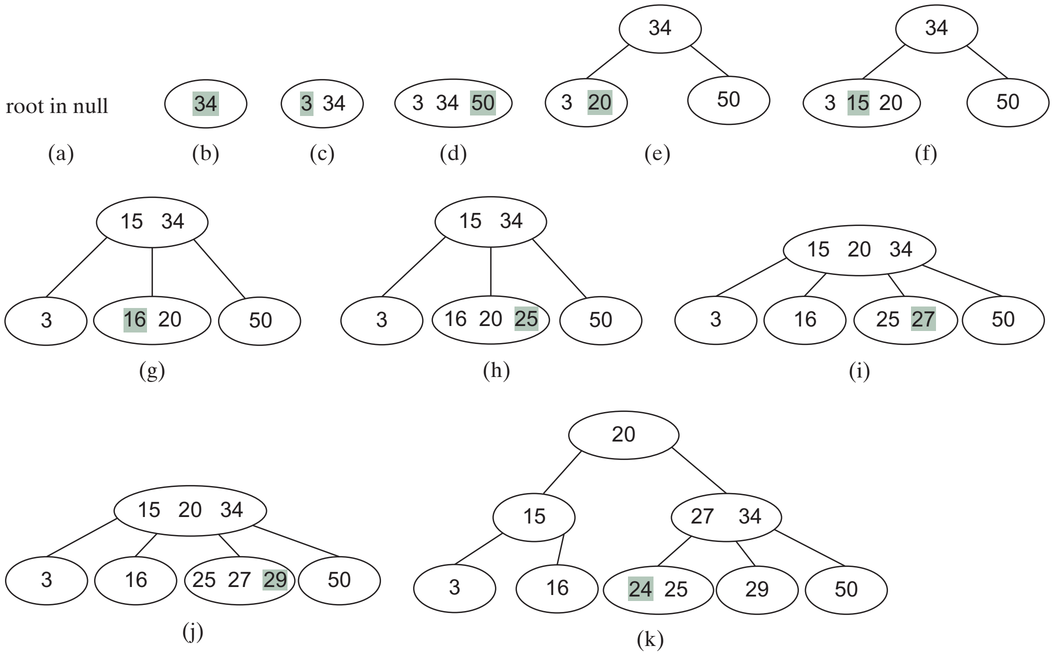
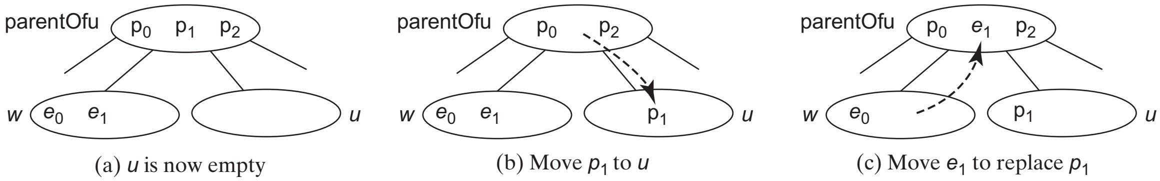


Figure 42.9 The tree changes after 34, 3, 50, 20, 15, 16, 25, 27, 29, and 24 are added into an empty tree.

Case 1: *u* is a leaf 3-node or 4-node. Delete *e* from *u*.

Case 2: *u* is a leaf 2-node. Delete *e* from *u*. Now *u* is empty. This situation is known as *underflow*. To remedy an underflow, consider two subcases:

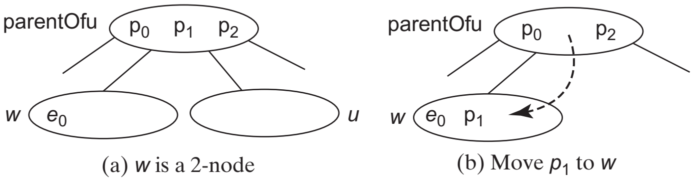
Case 2.1: *u*'s immediate left or right sibling is a 3- or 4-node. Let the node be *w*, as shown in Figure 42.10(a) (assume *w* is a left sibling of *u*). Perform a *transfer* operation that moves an element from *parentOfu* to *u*, as shown in Figure 42.10(b), and move an element from *w* to replace the moved element in *parentOfu*, as shown in Figure 42.10(c).



(a) *u* is now empty (b) Move p_1 to *u* (c) Move e_1 to replace p_1

Figure 42.10 The transfer operation fills the empty node *u*.

Case 2.2: Both *u*'s immediate left and right siblings are 2-node if they exist (*u* may have only one sibling). Let the node be *w*, as shown in Figure 42.11(a) (assume *w* is a left sibling of *u*). Perform a *fusion* operation that discards *u* and moves an element from *parentOfu* to *w*, as shown in Figure 42.11(b). If *parentOfu* becomes empty, repeat Case 2 recursively to perform a transfer or a fusion on *parentOfu*.

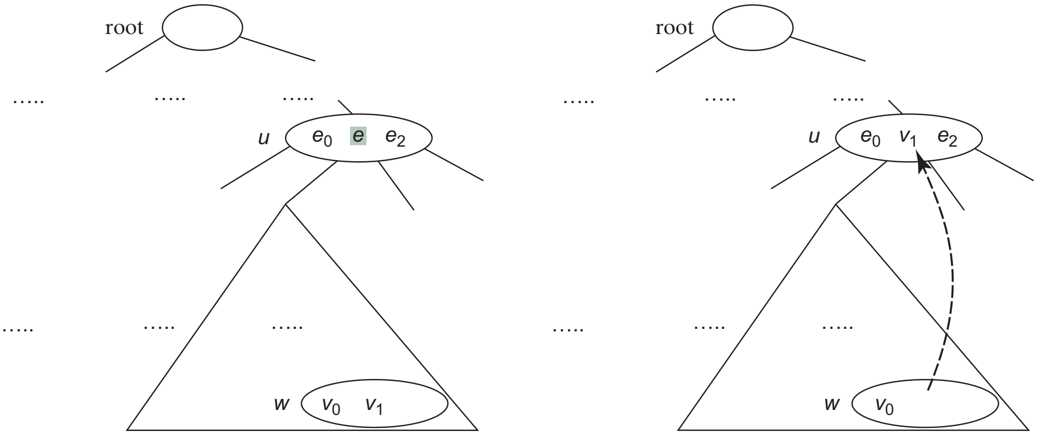


(a) *w* is a 2-node (b) Move p_1 to *w*

Figure 42.11 The fusion operation discards the empty node *u*.

Case 3: *u* is a nonleaf node. Find the rightmost leaf node in the left subtree of *e*. Let this node be *w*, as shown in Figure 42.12(a). Move the last element in *w* to replace *e* in *u*, as shown in Figure 42.12(b). If *w* becomes empty, apply a transfer or fusion operation on *w*.

Listing 42.3 describes the algorithm for deleting an element.



(a) *e* is in *u*

(b) Replace *e* with v_1

FIGURE 42.12 An element in the internal node is replaced by an element in a leaf node.

LISTING 42.3 Deleting an Element from a 2–4 tree

```
/** Delete the specified element from the tree */
public boolean delete(E e) {
  Locate the node that contains the element e
  if (the node is found) {
    delete(e, node); // Delete element e from the node
    size--; // After one element deleted
    return true; // Element deleted successfully
  }

  return false; // Element not in the tree
}

/** Delete the specified element from the node */
private void delete(E e, Tree24Node<E> node) {
  if (e is in a leaf node) {
    // Get the path that leads to e from the root
    ArrayList<Tree24Node<E>> path = path(e);

    Remove e from the node;

    // Check node for underflow along the path and fix it
    validate(e, node, path); // Check underflow node
  }
  else { // e is in an internal node
    Locate the rightmost node in the left subtree of node u;
    Get the rightmost element from the rightmost node;

    // Get the path that leads to e from the root
    ArrayList<Tree24Node<E>> path = path(rightmostElement);

    Replace the element in the node with the rightmost element

33      // Check node for underflow along the path and fix it
34      validate(rightmostElement, rightmostNode, path);
35    }
36  }
37
38  /** Perform a transfer or fusion operation if necessary */
39  private void validate(E e, Tree24Node<E> u,
40      ArrayList<Tree24Node<E>> path) {
41    for (int i = path.size() - 1; i >= 0; i--) {
42      if (u is not empty)
43        return; // Done, no need to perform transfer or fusion
44
45      Tree24Node<E> parentOfu = path.get(i - 1); // Get parent of u
46
47      // Check two siblings
48      if (left sibling of u has more than one element) {
49        Perform a transfer on u with its left sibling
50      }
51      else if (right sibling of u has more than one element) {
52        Perform a transfer on u with its right sibling
53      }
54      else if (u has left sibling) { // Fusion with a left sibling
55        Perform a fusion on u with its left sibling
56        u = parentOfu; // Back to the loop to check the parent node
57      }
58      else { // Fusion with right sibling (right sibling must exist)
59        Perform a fusion on u with its right sibling
60        u = parentOfu; // Back to the loop to check the parent node
61      }
62    }
63  }
```

The `delete(E e)` method locates the node that contains the element `e` and invokes the `delete(E e, Tree24Node<E> node)` method (line 5) to delete the element from the node.

If the node is a leaf node, get the path that leads to `e` from the root (line 17), delete `e` from the node (line 19), and invoke `validate` to check and fix the empty node (line 22). The `validate(E e, Tree24Node<E> u, ArrayList<Tree24Node<E>> path)` method performs a transfer or fusion operation if the node is empty. Since these operations may cause the parent of node `u` to become empty, a path is obtained in order to obtain the parents along the path from the root to node `u`, as shown in Figure 42.13.

If the node is a nonleaf node, locate the rightmost element in the left subtree of the node (lines 25–26), get the path that leads to the rightmost element from the root (line 29), replace

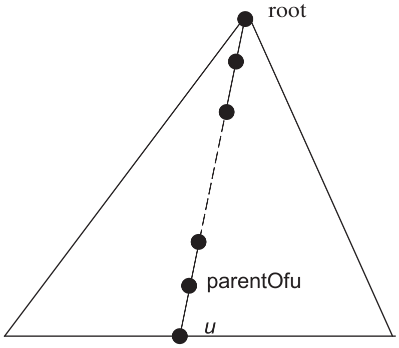


FIGURE 42.13 The nodes along the path may become empty as result of a transfer and fusion operation.

e in the node with the rightmost element (line 31), and invoke validate to fix the rightmost node if it is empty (line 34).

The validate(E e, Tree24Node<E> u, ArrayList<Tree24Node<E>> path) checks whether u is empty and performs a transfer or fusion operation to fix the empty node. The validate method exits when node is not empty (line 43). Otherwise, consider one of the following cases:

1. If u has a left sibling with more than one element, perform a transfer on u with its left sibling (line 49).
2. Otherwise, if u has a right sibling with more than one element, perform a transfer on u with its right sibling (line 52).
3. Otherwise, if u has a left sibling, perform a fusion on u with its left sibling (line 55) and reset u to parentOfu (line 56).
4. Otherwise, u must have a right sibling. Perform a fusion on u with its right sibling (line 59) and reset u to parentOfu (line 60).

Only one of the preceding cases is executed. Afterward, a new iteration starts to perform a transfer or fusion operation on a new node u if needed. Figure 42.14 shows the steps of deleting elements 20, 15, 3, 6, and 34 that are deleted from a 2–4 tree in Figure 42.9(k).

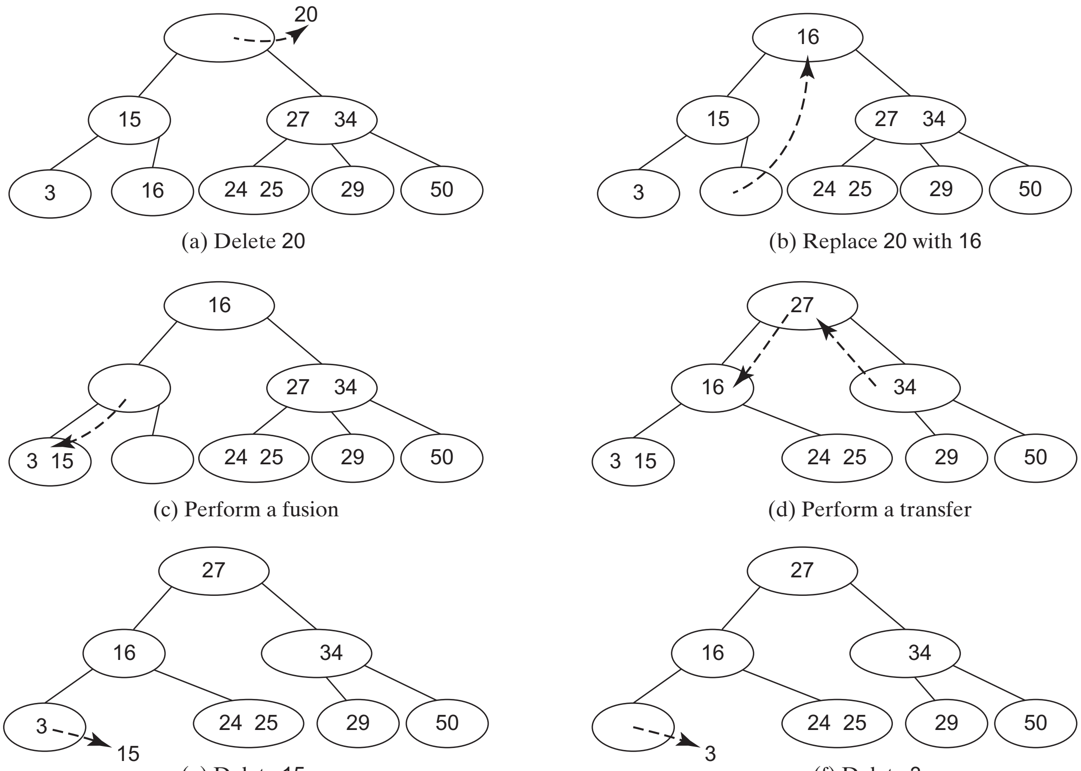
Check Point

42.5.1 How do you search an element in a 2–4 tree?

42.5.2 How do you insert an element into a 2–4 tree?

42.5.3 How do you delete an element from a 2–4 tree?

42.5.4 Show the change of a 2–4 tree when inserting 1, 2, 3, 4, 10, 9, 7, 5, 8, and 6 into it, in this order.



(a) Delete 20

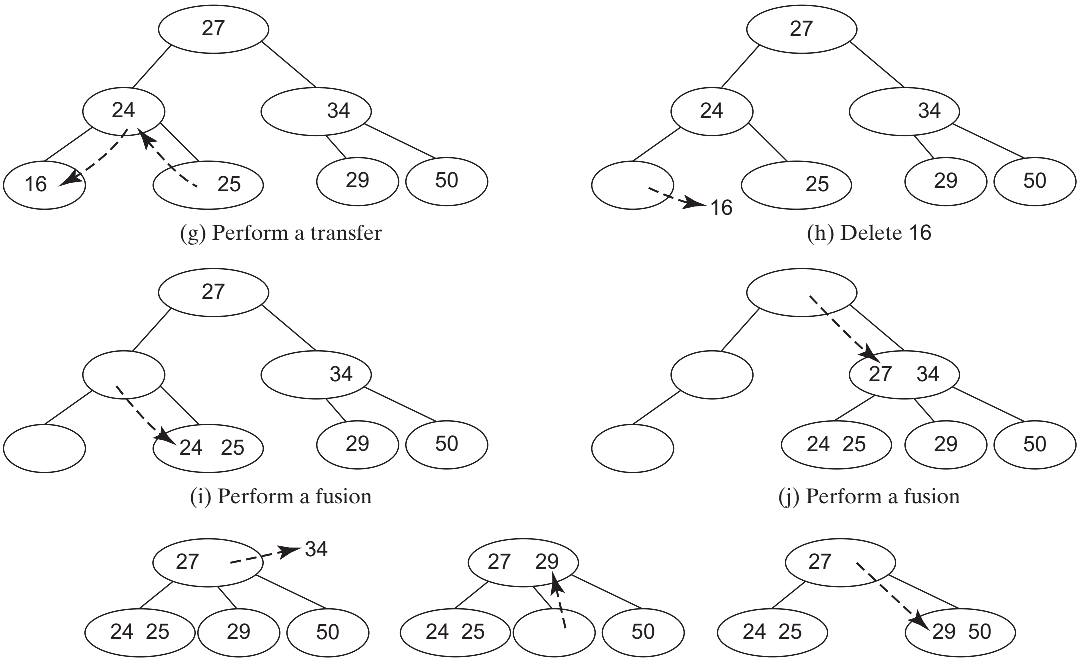
(b) Replace 20 with 16

(c) Perform a fusion

(d) Perform a transfer

(e) Delete 15

(f) Delete 3



(g) Perform a transfer

(h) Delete 16

(i) Perform a fusion

(j) Perform a fusion

(k) Delete 34

(l) Replace 34 with 16

(m) Perform a fusion

Figure 42.14 The tree changes after **20**, **15**, **3**, **6**, and **34** are deleted from a 2–4 tree.

42.5.5 For the tree built in the preceding question, show the change of the tree after deleting **1**, **2**, **3**, **4**, **10**, **9**, **7**, **5**, **8**, and **6** from it, in this order.

42.5.6 Show the change of a B-tree of order 6 when inserting **1**, **2**, **3**, **4**, **10**, **9**, **7**, **5**, **8**, **6**, **17**, **25**, **18**, **26**, **14**, **52**, **63**, **74**, **80**, **19**, and **27** into it, in this order.

42.5.7 For the tree built in the preceding question, show the change of the tree after deleting **1**, **2**, **3**, **4**, **10**, **9**, **7**, **5**, and **8**, and **6** from it, in this order.

42.6 Traversing Elements in a 2–4 tree

You can perform inorder, preorder, and postorder for traversing the elements in a 2–4 tree.

Inorder, preorder, and postorder traversals are useful for 2–4 trees. Inorder traversal visits the elements in increasing order. Preorder traversal visits the elements in the root, then recursively visits the subtrees from the left to right. Postorder traversal visits the subtrees from the left to right recursively, and then the elements in the root.

For example, in the 2–4 tree in Figure 42.9(k), the inorder traversal is
3 15 16 20 24 25 27 29 34 50

The preorder traversal is
20 15 3 16 27 34 24 25 29 50

The postorder traversal is
3 16 1 24 25 29 50 27 34 20

42.7 Implementing the **Tree24** Class

This section gives the complete implementation for the ***Tree24*** *class.*

Listing 42.4 gives the complete source code for the **Tree24** class.

Listing 42.4 Tree24.java

```java
import java.util.ArrayList;

public class Tree24<E extends Comparable<E>> implements Tree<E> {
  private Tree24Node<E> root;
  private int size;

  /** Create a default 2-4 tree */
  public Tree24() {
  }

  /** Create a 2-4 tree from an array of objects */
  public Tree24(E[] elements) {
    for (int i = 0; i < elements.length; i++)
      insert(elements[i]);
  }

  @Override /* Search an element in the tree */
  public boolean search(E e) {
    Tree24Node<E> current = root; // Start from the root

    while (current != null) {
      if (matched(e, current)) { // Element is in the node
        return true; // Element found
      }
      else {
        current = getChildNode(e, current); // Search in a subtree
      }
    }

    return false; // Element is not in the tree
  }

  /** Return true if the element is found in this node */
  private boolean matched(E e, Tree24Node<E> node) {
    for (int i = 0; i < node.elements.size(); i++)
      if (node.elements.get(i).equals(e))
        return true; // Element found

    return false; // No match in this node
  }

  /** Locate a child node to search element e */
  private Tree24Node<E> getChildNode(E e, Tree24Node<E> node) {
    if (node.child.size() == 0)
      return null; // node is a leaf

    int i = locate(e, node); // Locate the insertion point for e
    return node.child.get(i); // Return the child node
  }

  @Override /** Insert element e into the tree
   * Return true if the element is inserted successfully
   */
  public boolean insert(E e) {
    if (root == null)
      root = new Tree24Node<E>(e); // Create a new root for element
    else {
      // Locate the leaf node for inserting e
      Tree24Node<E> leafNode = null;
      Tree24Node<E> current = root;
      while (current != null)
        if (matched(e, current)) {
          return false; // Duplicate element found, nothing inserted
        }
        else {
          leafNode = current;
          current = getChildNode(e, current);
        }

      // Insert the element e into the leaf node
      insert(e, null, leafNode); // The right child of e is null
    }

    size++; // Increase size
    return true; // Element inserted
  }

  /** Insert element e into node u */
  private void insert(E e, Tree24Node<E> rightChildOfe,
      Tree24Node<E> u) {
    // Get the search path that leads to element e
    ArrayList<Tree24Node<E>> path = path(e);

    for (int i = path.size() - 1; i >= 0; i--) {
      if (u.elements.size() < 3) { // u is a 2-node or 3-node
        insert23(e, rightChildOfe, u); // Insert e to node u
        break; // No further insertion to u's parent needed
      }
      else {
        Tree24Node<E> v = new Tree24Node<E>(); // Create a new node
        E median = split(e, rightChildOfe, u, v); // Split u

        if (u == root) {
          root = new Tree24Node<E>(median); // New root
          root.child.add(u); // u is the left child of median
          root.child.add(v); // v is the right child of median
          break; // No further insertion to u's parent needed
        }
        else {
          // Use new values for the next iteration in the for loop
          e = median; // Element to be inserted to parent
          rightChildOfe = v; // Right child of the element
          u = path.get(i - 1); // New node to insert element
        }
      }
    }
  }

  /** Insert element to a 2- or 3- and return the insertion point */
  private void insert23(E e, Tree24Node<E> rightChildOfe,
      Tree24Node<E> node) {
    int i = this.locate(e, node); // Locate where to insert
    node.elements.add(i, e); // Insert the element into the node
    if (rightChildOfe != null)
      node.child.add(i + 1, rightChildOfe); // Insert the child link
  }

  /** Split a 4-node u into u and v and insert e to u or v */
  private E split(E e, Tree24Node<E> rightChildOfe,
      Tree24Node<E> u, Tree24Node<E> v) {
    // Move the last element in node u to node v
    v.elements.add(u.elements.remove(2));
    E median = u.elements.remove(1);

    // Split children for a nonleaf node
    // Move the last two children in node u to node v
    if (u.child.size() = 0) {
      v.child.add(u.child.remove(2));
      v.child.add(u.child.remove(2));
    }

    // Insert e into a 2- or 3- node u or v.
    if (e.compareTo(median) < 0)
      insert23(e, rightChildOfe, u);
    else
      insert23(e, rightChildOfe, v);

    return median; // Return the median element
  }

  /** Return a search path that leads to element e */
  private ArrayList<Tree24Node<E>= path(E e) {
    ArrayList<Tree24Node<E>= list = new ArrayList<Tree24Node<E>=();
    Tree24Node<E> current = root; // Start from the root

    while (current != null) {
      list.add(current); // Add the node to the list
      if (matched(e, current)) {
        break; // Element found
      }
      else {
        current = getChildNode(e, current);
      }
    }

    return list; // Return an array of nodes
  }

  @Override /** Delete the specified element from the tree */
  public boolean delete(E e) {
    // Locate the node that contains the element e
    Tree24Node<E> node = root;
    while (node != null)
      if (matched(e, node)) {
        delete(e, node); // Delete element e from node
        size--; // After one element deleted
        return true; // Element deleted successfully
      }
      else {
        node = getChildNode(e, node);
      }

    return false; // Element not in the tree
  }

  /** Delete the specified element from the node */
  private void delete(E e, Tree24Node<E> node) {
    if (node.child.size() == 0) { // e is in a leaf node
      // Get the path that leads to e from the root
      ArrayList<Tree24Node<E>> path = path(e);
      node.elements.remove(e); // Remove element e

      if (node == root) { // Special case
        if (node.elements.size() == 0)
          root = null; // Empty tree
        return; // Done
      }

      validate(e, node, path); // Check underflow node
    }
    else { // e is in an internal node
      // Locate the rightmost node in the left subtree of the node
      int index = locate(e, node); // Index of e in node
      Tree24Node<E> current = node.child.get(index);
      while (current.child.size() > 0) {
        current = current.child.get(current.child.size() - 1);
      }
      E rightmostElement =
        current.elements.get(current.elements.size() - 1);

      // Get the path that leads to e from the root
      ArrayList<Tree24Node<E>= path = path(rightmostElement);

      // Replace the deleted element with the rightmost element
      node.elements.set(index, current.elements.remove(
        current.elements.size() - 1));

      validate(rightmostElement, current, path); // Check underflow
    }
  }

  /** Perform transfer and confusion operations if necessary */
  private void validate(E e, Tree24Node<E> u,
      ArrayList<Tree24Node<E>> path) {
    for (int i = path.size() - 1; u.elements.size() == 0; i--) {
      Tree24Node<E> parentOfu = path.get(i - 1); // Get parent of u
      int k = locate(e, parentOfu); // Index of e in the parent node

      // Check two siblings
      if (k > 0 && parentOfu.child.get(k - 1).elements.size() > 1) {
        leftSiblingTransfer(k, u, parentOfu);
      }
      else if (k + 1 < parentOfu.child.size() &&
          parentOfu.child.get(k + 1).elements.size() > 1) {
        rightSiblingTransfer(k, u, parentOfu);
      }
      else if (k - 1 == 0) { // Fusion with a left sibling
        // Get left sibling of node u
        Tree24Node<E> leftNode = parentOfu.child.get(k - 1);

        // Perform a fusion with left sibling on node u
        leftSiblingFusion(k, leftNode, u, parentOfu);

        // Done when root becomes empty
        if (parentOfu == root && parentOfu.elements.size() == 0) {
          root = leftNode;
          break;
        }

        u = parentOfu; // Back to the loop to check the parent node
    }
    else { // Fusion with right sibling (right sibling must exist)
      // Get left sibling of node u
      Tree24Node<E> rightNode = parentOfu.child.get(k + 1);

      // Perform a fusion with right sibling on node u
      rightSiblingFusion(k, rightNode, u, parentOfu);

      // Done when root becomes empty
      if (parentOfu == root && parentOfu.elements.size() == 0) {
        root = rightNode;
        break;
      }

      u = parentOfu; // Back to the loop to check the parent node
    }
  }
}

/** Locate the insertion point of the element in the node */
private int locate(E o, Tree24Node<E> node) {
  for (int i = 0; i < node.elements.size(); i++) {
    if (o.compareTo(node.elements.get(i)) <= 0) {
      return i;
    }
  }

  return node.elements.size();
}

/** Perform a transfer with a left sibling */
private void leftSiblingTransfer(int k,
    Tree24Node<E> u, Tree24Node<E> parentOfu) {
  // Move an element from the parent to u
  u.elements.add(0, parentOfu.elements.get(k - 1));

  // Move an element from the left node to the parent
  Tree24Node<E> leftNode = parentOfu.child.get(k - 1);
  parentOfu.elements.set(k - 1,
    leftNode.elements.remove(leftNode.elements.size() - 1));

  // Move the child link from left sibling to the node
  if (leftNode.child.size() > 0)
    u.child.add(0, leftNode.child.remove(
      leftNode.child.size() - 1));
}

/** Perform a transfer with a right sibling */
private void rightSiblingTransfer(int k,
    Tree24Node<E> u, Tree24Node<E> parentOfu) {
  // Transfer an element from the parent to u
  u.elements.add(parentOfu.elements.get(k));

  // Transfer an element from the right node to the parent
  Tree24Node<E> rightNode = parentOfu.child.get(k + 1);
  parentOfu.elements.set(k, rightNode.elements.remove(0));

  // Move the child link from right sibling to the node
  if (rightNode.child.size() > 0)
    u.child.add(rightNode.child.remove(0));
}
```

```java

  /** Perform a fusion with a left sibling */
  private void leftSiblingFusion(int k, Tree24Node<E> leftNode,
      Tree24Node<E> u, Tree24Node<E> parentOfu) {
    // Transfer an element from the parent to the left sibling
    leftNode.elements.add(parentOfu.elements.remove(k - 1));

    // Remove the link to the empty node
    parentOfu.child.remove(k);

    // Adjust child links for nonleaf node
    if (u.child.size() > 0)
      leftNode.child.add(u.child.remove(0));
  }

  /** Perform a fusion with a right sibling */
  private void rightSiblingFusion(int k, Tree24Node<E> rightNode,
      Tree24Node<E> u, Tree24Node<E> parentOfu) {
    // Transfer an element from the parent to the right sibling
    rightNode.elements.add(0, parentOfu.elements.remove(k));

    // Remove the link to the empty node
    parentOfu.child.remove(k);

    // Adjust child links for nonleaf node
    if (u.child.size() > 0)
      rightNode.child.add(0, u.child.remove(0));
  }

  /** Get the number of nodes in the tree */
  public int getSize() {
    return size;
  }

  /** Preorder traversal from the root */
  public void preorder() {
    preorder(root);
  }

  /** Preorder traversal from a subtree */
  private void preorder(Tree24Node<E> root) {
    if (root == null)return;
    for (int i = 0; i < root.elements.size(); i++)
      System.out.print(root.elements.get(i) + " ");

    for (int i = 0; i < root.child.size(); i++)
      preorder(root.child.get(i));
  }

  /** Inorder traversal from the root*/
  public void inorder() {
    // Left as exercise
  }

  /** Postorder traversal from the root */
  public void postorder() {
    // Left as exercise
  }

  /** Return true if the tree is empty */
  public boolean isEmpty() {
```

```
364       return root == null;
365     }
366 
367     @Override /** Remove all elements from the tree */
368     public void clear() {
369       root = null;
370       size = 0;
371     }
372 
373     /** Return an iterator to traverse elements in the tree */
374     public java.util.Iterator iterator() {
375       // Left as exercise
376       return null;
377     }
378 
379     /** Define a 2-4 tree node */
380     protected static class Tree24Node<E extends Comparable<E>> {
381       // elements has maximum three values
382       ArrayList<E> elements = new ArrayList<E>(3);
383       // Each has maximum four childres
384       ArrayList<Tree24Node<E>> child
385         = new ArrayList<Tree24Node<E>>(4);
386 
387       /** Create an empty Tree24 node */
388       Tree24Node() {
389       }
390 
391       /** Create a Tree24 node with an initial element */
392       Tree24Node(E o) {
393         elements.add(o);
394       }
395     }
396   }
```

The Tree24 class contains the data fields root and size (lines 4–5). root references the root node and size stores the number of elements in the tree.

The Tree24 class has two constructors: a no-arg constructor (lines 8–9) that constructs an empty tree and a constructor that creates an initial Tree24 from an array of elements (lines 12–15).

The search method (lines 18–31) searches an element in the tree. It returns true (line 23) if the element is in the tree and returns false if the search arrives at an empty subtree (line 30).

The matched(e, node) method (lines 34–40) checks where the element e is in the node.

The getChildNode(e, node) method (lines 43–49) returns the root of a subtree where e should be searched.

The insert(E e) method inserts an element in a tree (lines 54–76). If the tree is empty, a new root is created (line 56). The method locates a leaf node in which the element will be inserted and invokes insert(e, null, leafNode) to insert the element (line 71).

The insert(e, rightChildOfe, u) method inserts an element into node u (lines 79–107). The method first invokes path(e) (line 82) to obtain a search path from the root to node u. Each iteration of the for loop considers u and its parent parentOfu (lines 84–106). If u is a 2-node or 3-node, invoke insert23(e, rightChildOfe, u) to insert e and its child link rightChildOfe into u (line 86). No split is needed (line 87). Otherwise, create a new node v (line 90) and invoke split(e, rightChildOfe, u, v) (line 91) to split u into u and v. The split method inserts e into either u and v and returns the median in the original u. If u is the root, create a new root to hold median, and set u and v as the left and right children for median (lines 95–96). If u is not the root, insert median to parentOfu in the next iteration (lines 101–103).

The insert23(e, rightChildOfe, node) method inserts e along with the reference to its right child into the node (lines 110–116). The method first invokes locate(e, node) (line 112) to locate an insertion point, then insert e into the node (line 113). If rightChildOfe is not null, it is inserted into the child list of the node (line 115).

The split(e, rightChildOfe, u, v) method splits a 4-node u (lines 119-139). This is accomplished as follows: (1) move the last element from u to v and remove the median element from u (lines 122–123); (2) move the last two child links from u to v (lines 127–130) if u is a nonleaf node; (3) if e < median, insert e into u; otherwise, insert e into v (lines 133–136); and (4) return median (line 138).

The path(e) method returns an ArrayList of nodes searched from the root in order to locate e (lines 142–157). If e is in the tree, the last node in the path contains e. Otherwise, the last node is where e should be inserted.

The delete(E e) method deletes an element from the tree (lines 160–174). The method first locates the node that contains e and invokes delete(e, node) to delete e from the node (line 165). If the element is not in the tree, return false (line 173).

The delete(e, node) method deletes an element from node u (lines 177–211). If the node is a leaf node, obtain the path that leads to e (line 180), delete e (line 182), set root to null if the tree becomes empty (lines 184–188), and invoke validate to apply transfer and fusion operation on empty nodes (line 190). If the node is a nonleaf node, locate the rightmost element (lines 194–200), obtain the path that leads to e (line 203), replace e with the rightmost element (lines 206–207), and invoke validate to apply transfer and fusion operations on empty nodes (line 209).

The validate(e, u, path) method ensures that the tree is a valid 2–4 tree (lines 214–259). The for loop terminates when u is not empty (line 216). The loop body is executed to fix the empty node u by performing a transfer or fusion operation. If a left sibling with more than one element exists, perform a transfer on u with the left sibling (line 222). Otherwise, if a right sibling with more than one element exists, perform a transfer on u with the left sibling (line 226). Otherwise, if a left sibling exists, perform a fusion on u with the left sibling (lines 230–239), and validate parentOfu in the next loop iteration (line 241). Otherwise, perform a fusion on u with the right sibling.

The locate(e, node) method locates the index of e in the node (lines 262–270).

The leftSiblingTransfer(k, u, parentOfu) method performs a transfer on u with its left sibling (lines 273–287). The rightSiblingTransfer(k, u, parentOfu) method performs a transfer on u with its right sibling (lines 290–302). The leftSiblingFusion(k, leftNode, u, parentOfu) method performs a fusion on u with its left sibling leftNode (lines 305–316). The rightSiblingFusion(k, rightNode, u, parentOfu) method performs a fusion on u with its right sibling rightNode (lines 319–330).

The preorder() method displays all the elements in the tree in preorder (lines 338–350).

The inner class Tree24Node defines a class for a node in the tree (lines 374–389).

42.8 Testing the Tree24 Class

*This section writes a test program for using the **Tree24** class.*

Key Point

Listing 42.5 gives a test program. The program creates a 2–4 tree and inserts elements in lines 6–20, and deletes elements in lines 22–56.

Listing 42.5 TestTree24.java

```
1  public class TestTree24 {
2    public static void main(String[] args) {
3      // Create a 2-4 tree
4      Tree24<Integer> tree = new Tree24<Integer>();
5
```

```
    tree.insert(34);
    tree.insert(3);
    tree.insert(50);
    tree.insert(20);
    tree.insert(15);
    tree.insert(16);
    tree.insert(25);
    tree.insert(27);
    tree.insert(29);
    tree.insert(24);
    System.out.print("\nAfter inserting 24:");
    printTree(tree);
    tree.insert(23);
    tree.insert(22);
    tree.insert(60);
    tree.insert(70);
    System.out.print("\nAfter inserting 70:");
    printTree(tree);

    tree.delete(34);
    System.out.print("\nAfter deleting 34:");
    printTree(tree);

    tree.delete(25);
    System.out.print("\nAfter deleting 25:");
    printTree(tree);

    tree.delete(50);
    System.out.print("\nAfter deleting 50:");
    printTree(tree);

    tree.delete(16);
    System.out.print("\nAfter deleting 16:");
    printTree(tree);

    tree.delete(3);
    System.out.print("\nAfter deleting 3:");
    printTree(tree);

    tree.delete(15);
    System.out.print("\nAfter deleting 15:");
    printTree(tree);
  }

  public static <E extends Comparable<E>>
    void printTree(Tree<E> tree) {
    // Traverse tree
    System.out.print("\nPreorder: ");
    tree.preorder();
    System.out.print("\nThe number of nodes is " + tree.getSize());
    System.out.println();
  }
}
```

```
After inserting 24:
Preorder: 20 15 3 16 27 34 24 25 29 50
The number of nodes is 10

After inserting 70:
Preorder: 20 15 3 16 24 27 34 22 23 25 29 50 60 70
The number of nodes is 14

After deleting 34:
Preorder: 20 15 3 16 24 27 50 22 23 25 29 60 70
The number of nodes is 13

After deleting 25:
Preorder: 20 15 3 16 23 27 50 22 24 29 60 70
The number of nodes is 12

After deleting 50:
Preorder: 20 15 3 16 23 27 60 22 24 29 70
The number of nodes is 11

After deleting 16:
Preorder: 23 20 3 15 22 27 60 24 29 70
The number of nodes is 10

After deleting 3:
Preorder: 23 20 15 22 27 60 24 29 70
The number of nodes is 9

After deleting 15:
Preorder: 27 23 20 22 24 60 29 70
The number of nodes is 8
```

Figure 42.15 shows how the tree evolves as elements are added. After **34**, **3**, **50**, **20**, **15**, **16**, **25**, **27**, **29**, and **24** are added to the tree, it is as shown in Figure 42.15(a). After inserting **23**, **22**, **60**, and **70**, the tree is as shown in Figure 42.15(b). After deleting **34**, the tree is as shown in Figure 42.15(c). After deleting **25**, the tree is as shown in Figure 42.15(d). After deleting **50**, the tree is as shown in Figure 42.15(e). After deleting **16**, the tree is as shown in Figure 42.15(f). After deleting **3**, the tree is as shown in Figure 42.15(g). After deleting **15**, the tree is as shown in Figure 42.15(h).

42.9 Time-Complexity Analysis

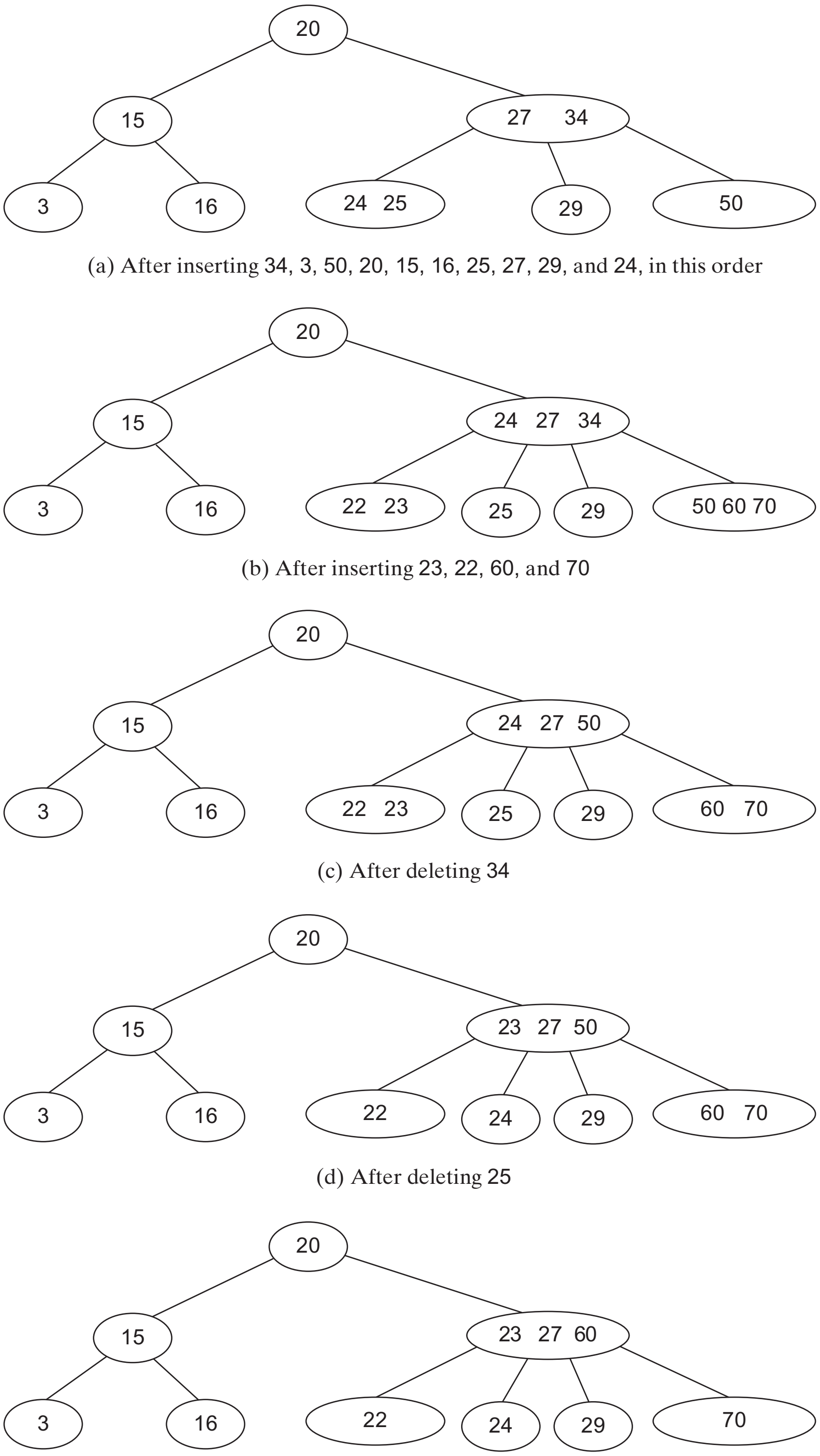
Search, insertion, and deletion operations take O(logn) time in a 2–4 tree.

Since a 2–4 tree is a completely balanced binary tree, its height is at most $O(\log n)$. The **search**, **insert**, and **delete** methods operate on the nodes along a path in the tree. It takes a constant time to search an element within a node. So, the **search** method takes $O(\log n)$ time. For the **insert** method, the time for splitting a node takes a constant time. So, the **insert** method takes $O(\log n)$ time. For the **delete** method, it takes a constant time to perform a transfer and fusion operation. So, the **delete** method takes $O(\log n)$ time.

42.10 B-Tree

A B-tree is a generalization of a 2–4 tree.

So far we assume the entire data set is stored in main memory. What if the data set is too large and cannot fit in the main memory, as in the case with most databases, where data is stored on disks? Suppose you use an AVL tree to organize a million records in a database table. To find a record, the average number of nodes traversed is $\log_2 1{,}000{,}000 \approx 20$. This is fine if all nodes are stored in main memory. However, for nodes stored on a disk, this means 20 disk reads. Disk I/O is expensive, and it is thousands of times slower than memory access. To improve performance, we need to reduce the number of disk I/Os. An efficient data structure for performing search, insertion, and deletion for data stored on secondary storage such as hard disks is the B-tree, which is a generalization of the 2–4 tree.



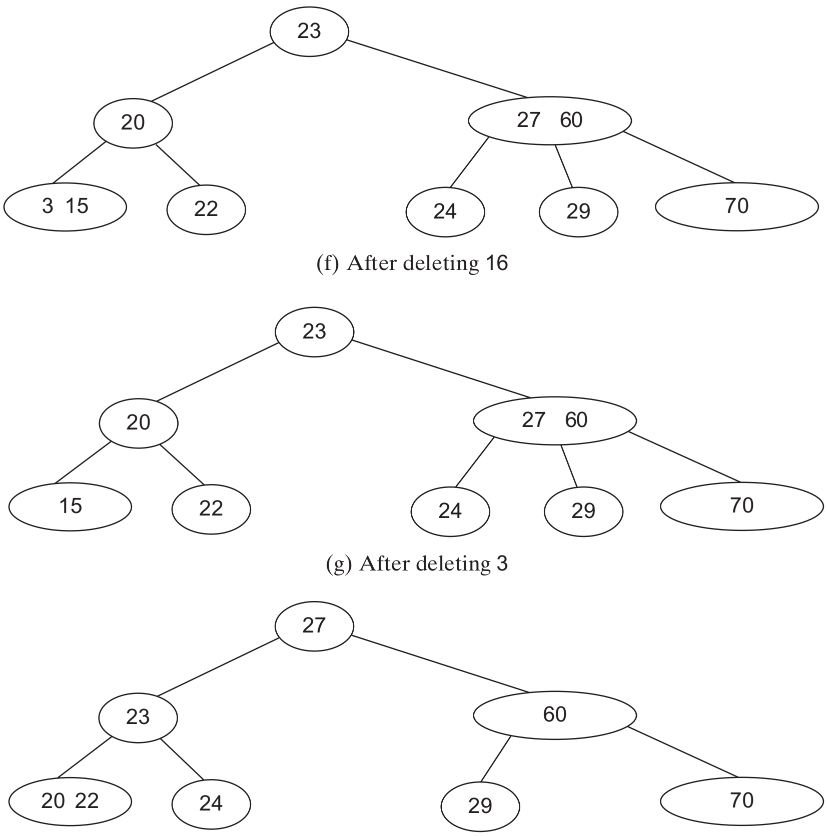
(a) After inserting 34, 3, 50, 20, 15, 16, 25, 27, 29, and 24, in this order

(b) After inserting 23, 22, 60, and 70

(c) After deleting 34

(d) After deleting 25

(e) After deleting 50



(f) After deleting 16

(g) After deleting 3

(h) After deleting 15

Figure 42.15 The tree evolves as elements are inserted and deleted.

A B-tree of order *d* is defined as follows:

1. Each node except the root contains between $\lceil d/2 \rceil - 1$ and $d - 1$ keys.
2. The root may contain up to $d - 1$ keys.
3. A nonleaf node with k keys has $k + 1$ children.
4. All leaf nodes have the same depth.

Figure 42.16 shows a B-tree of order 6. For simplicity, we use integers to represent keys. Each key is associated with a pointer that points to the actual record in the database. For simplicity, the pointers to the records in the database are omitted in the figure.

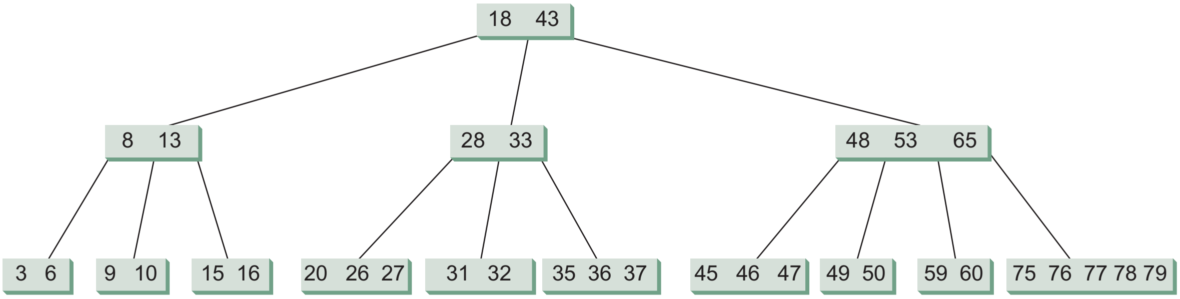


Figure 42.16 In a B-tree of order 6, each node except the root may contain between 2 and 5 keys.

Note that a B-tree is a search tree. The keys in each node are placed in increasing order. Each key in an interior node has a left subtree and a right subtree, as shown in Figure 42.17. All keys in the left subtree are less than the key in the parent node, and all keys in the right subtree are greater than the key in the parent node.

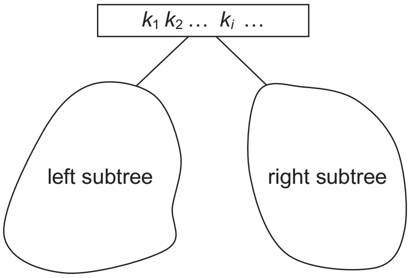


FIGURE 42.17 The keys in the left (right) subtree of key k_i are less than (greater than) k_i.

The basic unit of the IO operations on a disk is a block. When you read data from a disk, the whole block that contains the data is read. You should choose an appropriate order *d* so that a node can fit in a single disk block. This will minimize the number of disk IOs.

A 2–4 tree is actually a B-tree of order 4. The techniques for insertion and deletion in a 2–4 tree can be easily generalized for a B-tree.

Inserting a key to a B-tree is similar to what was done for a 2–4 tree. First, locate the leaf node in which the key will be inserted. Insert the key to the node. After the insertion, if the leaf node has *d* keys, an overflow occurs. To resolve overflow, perform a *split* operation similar to the one used in a 2–4 tree, as follows:

Let `u` denote the node needed to be split and let `m` denote the median key in the node. Create a new node and move all keys greater than `m` to this new node. Insert `m` to the parent node of `u`. Now `u` becomes the left child of `m` and `v` becomes the right child of `m`, as shown in Figure 42.18. If inserting `m` into the parent node of `u` causes an overflow, repeat the same split process on the parent node.

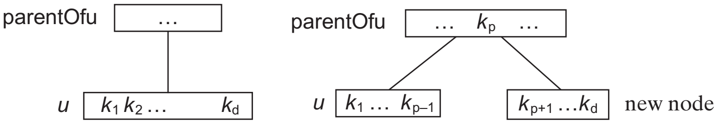
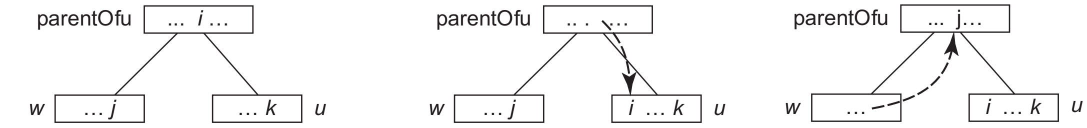


FIGURE 42.18 (a) After inserting a new key to node *u*. (b) The median key k_p is inserted to `parentOfu`.

A key *k* can be deleted from a B-tree in the same way as in a 2–4 tree. First locate the node *u* that contains the key. Consider two cases:

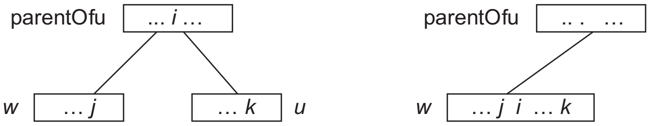
Case 1: If *u* is a leaf node, remove the key from *u*. After the removal, if *u* has less than $\lceil d/2 \rceil - 1$ keys, an underflow occurs. To remedy an underflow, perform a transfer with a sibling



(a) Before a transfer is performed (b) Key *i* moved to node *u* (c) Key *j* moved to parentOfu

FIGURE 42.19 The transfer operation transfers a key from the `parentOfu` to *u* and transfers a key from *u* 's sibling `parentOfu`.

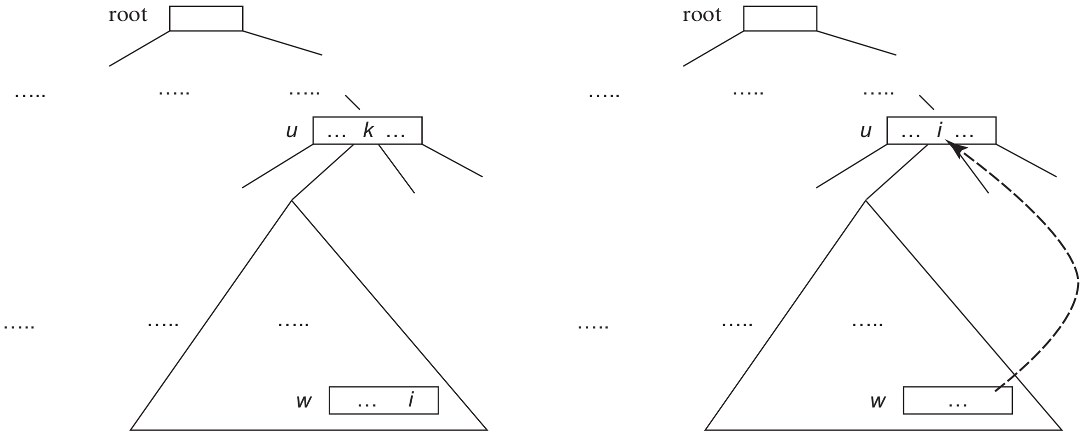
w of *u* that has more than $\lceil d/2 \rceil - 1$ keys if such sibling exists, as shown in Figure 42.19. Otherwise, perform a fusion with a sibling *w* of *u*, as shown in Figure 42.20.



(a) Before a fusion is performed (b) After a fusion is performed

FIGURE 42.20 The fusion operation moves key *i* from the `parentOfu` *u* to *w* and moves all keys in *u* to *w*.

Case 2: *u* is a nonleaf node. Find the rightmost leaf node in the left subtree of *k*. Let this node be *w*, as shown in Figure 42.21(a). Move the last key in *w* to replace *k* in *u*, as shown in Figure 42.21(b). If *w* becomes underflow, apply a transfer or fusion operation on *w*.



(a) Key is in *u* (b) Replace key *k* with key *i*

FIGURE 42.21 A key in the internal node is replaced by an element in a leaf node.

The performance of a B-tree depends on the number of disk IOs (i.e., the number of nodes accessed). The number of nodes accessed for search, insertion, and deletion operations depends on the height of the tree. In the worst case, each node contains $\lceil d/2 \rceil - 1$ keys. So, the height of the tree is $\log_{\lceil d/2 \rceil} n$, where *n* is the number of keys. In the best case, each node contains $d - 1$ keys. So, the height of the tree is $\log_d n$. Consider a B-tree of order 12 for 10 million keys. The height of the tree is between $\log_6 10{,}000{,}000 \approx 7$ and $\log_{12} 10{,}000{,}000 \approx 9$. So, for search, insertion, and deletion operations, the maximum number of nodes visited is 42. If you use an AVL tree, the maximum number of nodes visited is $\log_2 10{,}000{,}000 \approx 24$.

KEY TERMS

2–3–4 tree	42-2
2–4 tree	42-2
2-node	42-2
3-node	42-2
4-node	42-2
B-tree	42-11
fusion operation	42-7
split operation	42-4
transfer operation	42-7

Chapter Summary

1. A 2–4 tree is a completely balanced search tree. In a 2–4 tree, a node may have one, two, or three elements.

2. Searching an element in a 2–4 tree is similar to searching an element in a binary tree. The difference is that you have searched an element within a node.

3. To insert an element to a 2–4 tree, locate a leaf node in which the element will be inserted. If the leaf node is a 2- or 3-node, simply insert the element into the node. If the node is a 4-node, split the node.

4. The process of deleting an element from a 2–4 tree is similar to that of deleting an element from a binary tree. The difference is that you have to perform transfer or fusion operations for empty nodes.

5. The height of a 2–4 tree is $O(\log n)$. So, the time complexities for the search, insert, and delete methods are $O(\log n)$.

6. A B-tree is a generalization of the 2–4 tree. Each node in a B-tree of order d can have between $\lceil d/2 \rceil - 1$ and $d - 1$ keys except the root. 2–4 trees are flatter than AVL trees and B-trees are flatter than 2–4 trees. B-trees are efficient for creating indexes for data in database systems where large amounts of data are stored on disks.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

Programming Exercises

***42.1** (*Implement* `inorder`) The `inorder` method in `Tree24` is left as an exercise. Implement it.

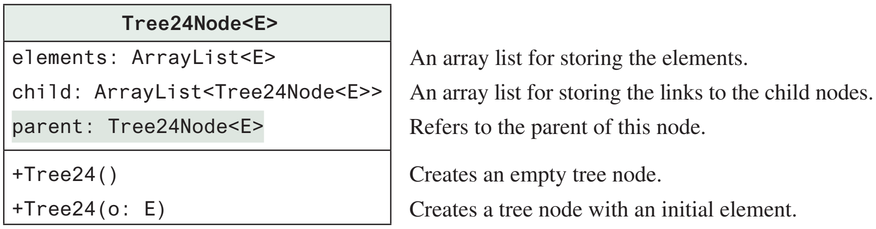
42.2 (*Implement* `postorder`) The `postorder` method in `Tree24` is left as an exercise. Implement it.

42.3 (*Implement* `iterator`) The `iterator` method in `Tree24` is left as an exercise. Implement it to iterate the elements using inorder.

***42.4** (*Display a 2–4 tree graphically*) Write a GUI program that displays a 2–4 tree.

*****42.5** (*2–4 tree animation*) Write a GUI program that animates the 2–4 tree `insert`, `delete`, and `search` methods, as shown in Figure 42.4.

****42.6** (*Parent reference for* `Tree24`) Redefine `Tree24Node` to add a reference to a node's parent, as shown below:



```
Tree24Node<E>

elements: ArrayList<E>               An array list for storing the elements.
child: ArrayList<Tree24Node<E>>      An array list for storing the links to the child nodes.
parent: Tree24Node<E>                Refers to the parent of this node.

+Tree24()                            Creates an empty tree node.
+Tree24(o: E)                        Creates a tree node with an initial element.
```

Add the following two new methods in `Tree24`:

```
public Tree24Node<E> getParent(Tree24Node<E> node)
  Returns the parent for the specified node.
public ArrayList<Tree24Node<E>> getPath(Tree24Node<E> node)
  Returns the path from the specified node to the root in an
array list.
```

Write a test program that adds numbers **1**, **2**, ..., **100** to the tree and displays the paths for all leaf nodes.

*****42.7** (*The* ***BTree*** *class*) Design and implement a class for B-trees.

CHAPTER

43

Red-Black Trees

Objectives

- To know what a red-black tree is (§43.1).
- To convert a red-black tree to a 2–4 tree and vice versa (§43.2).
- To design the `RBTree` class that extends the `BST` class (§43.3).
- To insert an element in a red-black tree and resolve the double-red violation if necessary (§43.4).
- To delete an element from a red-black tree and resolve the double-black problem if necessary (§43.5).
- To implement and test the `RBTree` class (§§43.6–43.7).
- To compare the performance of AVL trees, 2–4 trees, and `RBTree` (§43.8).

43.1 Introduction

Key Point

A red-black tree is a balanced binary search tree derived from a 2–4 tree. *A red-black tree corresponds to a 2-4 tree.*

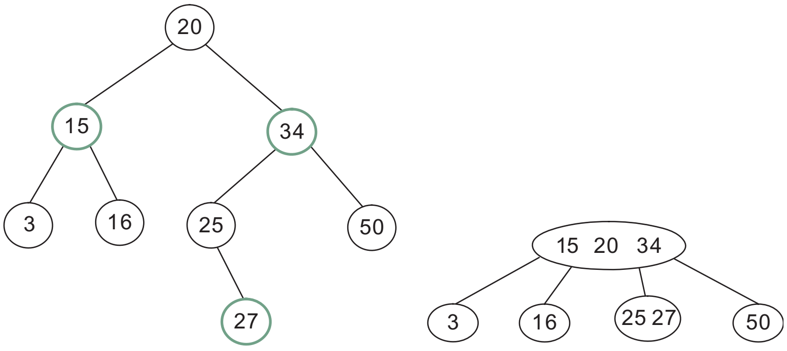
Each node in a red-black tree has a *color attribute* red or black, as shown in Figure 43.1(a). A node is called *external* if its left or right subtree is empty. Note that a leaf node is external, but an external node is not necessarily a leaf node. For example, node **25** is external, but it is not a leaf. The *black depth* of a node is defined as the number of black nodes in a path from the node to the root. For example, the black depth of node **25** is **2**, and that of node **27** is **2**.

Note
The red nodes appear in blue in the text.

A red-black tree has the following properties:

1. The root is black.
2. Two adjacent nodes cannot be both red.
3. All external nodes have the same black depth.

The red-black tree in Figure 43.1(a) satisfies all three properties. A red-black tree can be converted to a 2-4 tree, and vice versa. Figure 43.1(b) shows an equivalent 2-4 tree for the red-black tree in Figure 43.1(a).



(a) A red-black tree

(b) A 2-4 tree

FIGURE 43.1 A red-black tree can be represented using a 2-4 tree, and vice versa.

43.2 Conversion between Red-Black Trees and 2-4 Trees

Key Point

This section discusses the correspondence between a red-black tree and a 2-4 tree.

You can design insertion and deletion algorithms for red-black trees without having knowledge of 2-4 trees. However, the correspondence between red-black trees and 2-4 trees provides useful intuition about the structure of red-black trees and operations. For this reason, this section discusses the correspondence between these two types of trees.

To convert a red-black tree to a 2-4 tree, simply merge every red node with its parent to create a 3-node or a 4-node. For example, the red nodes **15** and **34** are merged to their parent to create a 4-node, and the red node **27** is merged to its parent to create a 3-node, as shown in Figure 43.1(b).

To convert a 2-4 tree to a red-black tree, perform the following transformations for each node *u*:

1. If *u* is a 2-node, color it black, as shown in Figure 43.2(a).
2. If *u* is a 3-node with element values e_0 and e_1, there are two ways to convert it. Either make e_0 the parent of e_1 or make e_1 the parent of e_0. In any case, color the parent black and the child red, as shown in Figure 43.2(b).
3. If *u* is a 4-node with element values e_0, e_1, and e_2, make e_1 the parent of e_0 and e_2. Color e_1 black and e_0 and e_2 red, as shown in Figure 43.2(c).

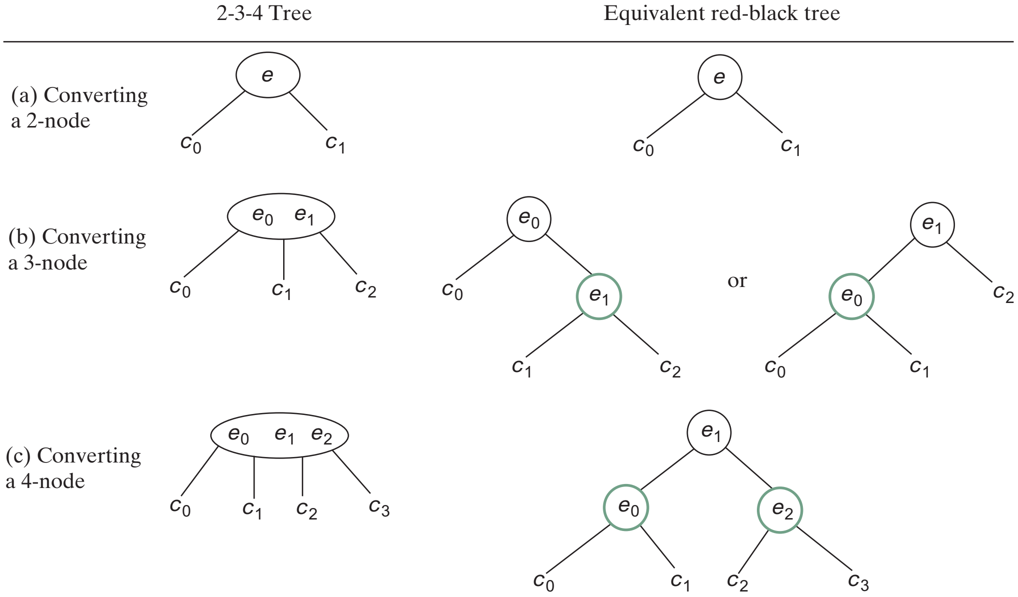
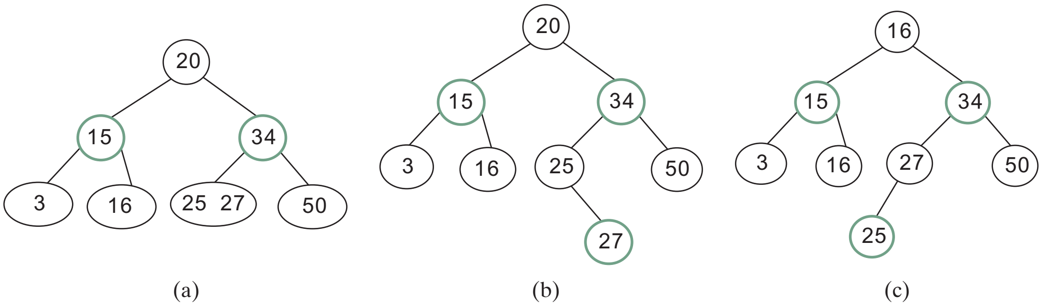


FIGURE 43.2 A node in a 2-4 tree can be transformed to nodes in a red-black tree.

Let us apply the transformation for the 2-4 tree in Figure 43.1(b). After transforming the 4-node, the tree is as shown in Figure 43.3(a). After transforming the 3-node, the tree is as shown in Figure 43.3(b). Note the transformation for a 3-node is not unique. Therefore, the conversion from a 2-4 tree to a red-black tree is *not unique*. After transforming the 3-node, the tree could also be as shown in Figure 43.3(c).



(a) (b) (c)

FIGURE 43.3 The conversion from a 2-4 tree to a red-black tree is not unique.

You can prove the conversion results in a red-black tree that satisfies all three properties.

Property 1. The root is black.

> Proof: If the root of a 2-4 tree is a 2-node, the root of the red-black tree is black. If the root of a 2-4 tree is a 3-node or 4-node, the transformation produces a black parent at the root.

Property 2. Two adjacent nodes cannot be both red.

> Proof: Since the parent of a red node is always black, no two adjacent nodes can be both red.

Property 3. All external nodes have the same black depth.

> Proof: When you covert a node in a 2-4 tree to red-black tree nodes, you get one black node and zero, one, or two red nodes as its children, depending on whether the original node is a 2-, 3-, or 4-node. Only a leaf 2-4 node may produce external red-black nodes. Since a 2-4 tree is perfectly balanced, the number of black nodes in any path from the root to an external node is the same.

Check Point

43.2.1 What is a red-black tree? What is an external node? What is black depth?

43.2.2 Describe the properties of a red-black tree.

43.2.3 How do you convert a red-black tree to a 2-4 tree? Is the conversion unique?

43.2.4 How do you convert a 2-4 tree to a red-black tree? Is the conversion unique?

43.3 Designing Classes for Red-Black Trees

Key Point

A red-black tree designs a class for a red-black tree.

A red-black tree is a binary search tree. So, you can define the `RBTree` class to extend the `BST` class, as shown in Figure 43.4. The `BST` and `TreeNode` classes are defined in §26.2.5.

Each node in a red-black tree has a color property. Because the color is either red or black, it is efficient to use the `boolean` type to denote it. The `RBTreeNode` class can be defined to extend `BST.TreeNode` with the color property. For convenience, we also provide the methods for checking the color and setting a new color. Note that `TreeNode` is defined as a static inner class in `BST`. `RBTreeNode` will be defined as a static inner class in `RBTree`. Note that `BSTNode` contains the data fields `element`, `left`, and `right`, which are inherited in `RBTreeNode`. So, `RBTreeNode` contains four data fields, as pictured in Figure 43.5.

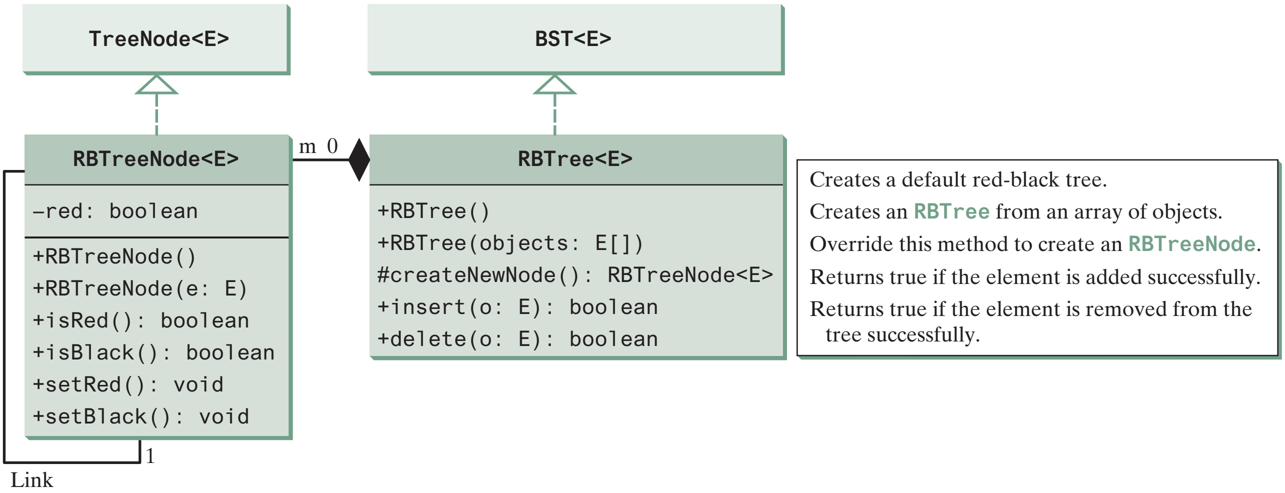
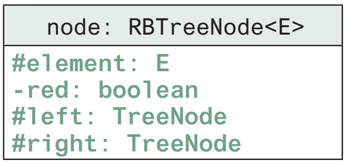

TreeNode<E>
BST<E>
RBTreeNode<E>
-red: boolean
+RBTreeNode()
+RBTreeNode(e: E)
+isRed(): boolean
+isBlack(): boolean
+setRed(): void
+setBlack(): void
m 0
1
Link
RBTree<E>
+RBTree()
+RBTree(objects: E[])
#createNewNode(): RBTreeNode<E>
+insert(o: E): boolean
+delete(o: E): boolean
Creates a default red-black tree.
Creates an RBTree from an array of objects.
Override this method to create an RBTreeNode.
Returns true if the element is added successfully.
Returns true if the element is removed from the tree successfully.

Figure 43.4 The `RBTree` class extends `BST` with new implementations for the `insert` and `delete` methods.


node: RBTreeNode<E>

#element: E
-red: boolean
#left: TreeNode
#right: TreeNode

FIGURE 43.5 An RBTreeNode contains data fields element, red, left, and right.

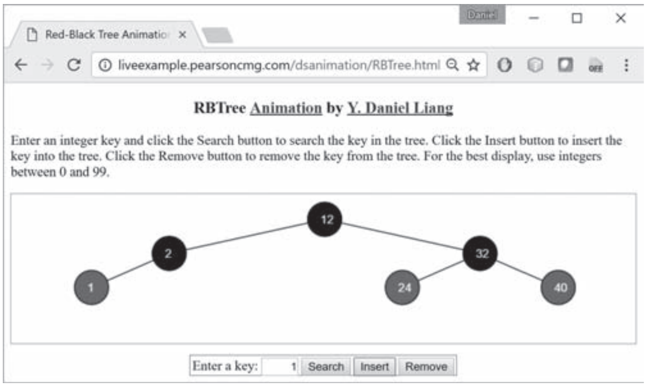
In the BST class, the createNewNode() method creates a TreeNode object. This method is overridden in the RBTree class to create an RBTreeNode. Note the return type of the createNewNode() method in the BST class is TreeNode, but the return type of the createNewNode() method in RBTree class is RBTreeNode. This is fine, since RBTreeNode is a subtype of TreeNode.

Searching an element in a red-black tree is the same as searching in a regular binary search tree. So, the search method defined in the BST class also works for RBTree.

The insert and delete methods are overridden to insert and delete an element and perform operations for coloring and restructuring if necessary to ensure that the three properties of the red-black tree are satisfied.

Pedagogical Note

Run from http://liveexample.pearsoncmg.com/dsanimation/RBTree.html to see how a red-black tree works, as shown in Figure 43.6.



RBTree Animation by Y. Daniel Liang

Enter an integer key and click the Search button to search the key in the tree. Click the Insert button to insert the key into the tree. Click the Remove button to remove the key from the tree. For the best display, use integers between 0 and 99.

FIGURE 43.6 The animation tool enables you to insert, delete, and search elements in a red-black tree visually.

43.4 Overriding the insert Method

This section discusses how to insert an element to red-black tree.

Key Point

A new element is always inserted as a leaf node. If the new node is the root, color it black. Otherwise, color it red. If the parent of the new node is red, it violates Property 2 of the red-black tree. We call this a *double-red* violation.

Let u denote the new node inserted, v the parent of u, w the parent of v, and x the sibling of v. To fix the double-red violation, consider two cases:

Case 1: x is black or x is null. There are four possible configurations for u, v, w, and x, as shown in Figures 43.7(a), 43.8(a), 43.9(a), and 43.10(a). In this case, u, v, and w form a 4-node in the corresponding 2-4 tree, as shown in Figures 43.7(c), 43.8(c), 43.9(c), and 43.10(c), but are represented incorrectly in the red-black tree. To correct this error, restructure and recolor three nodes u, v, and w, as shown in Figures 43.7(b), 43.8(b), 43.9(b), and 43.10(b). Note x, y_1, y_2, and y_3 may be null.

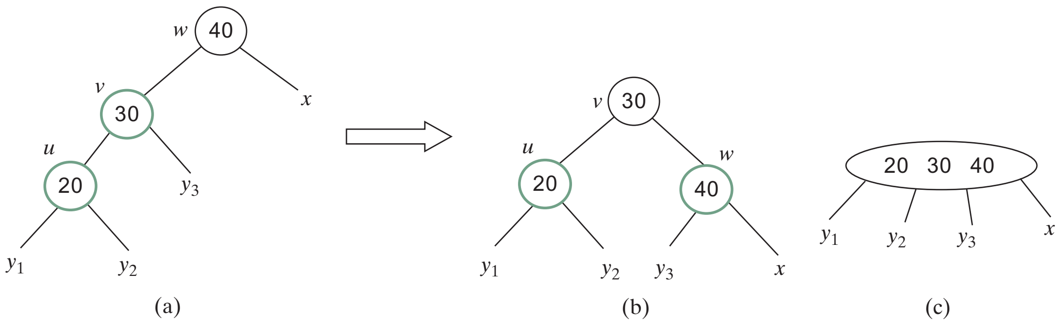


FIGURE 43.7 Case 1.1: $u < v < w$.

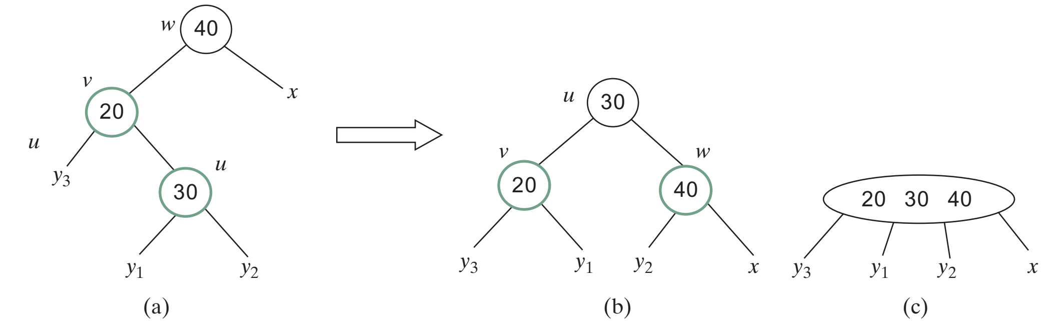


FIGURE 43.8 Case 1.2: $v < u < w$

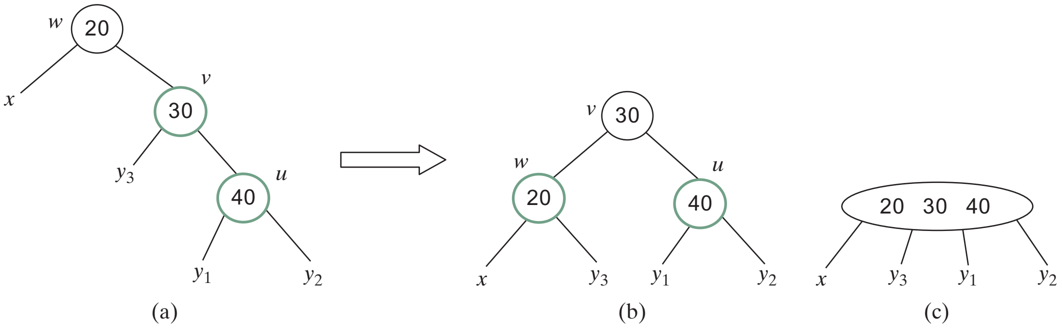


FIGURE 43.9 Case 1.3: $w < v < u$

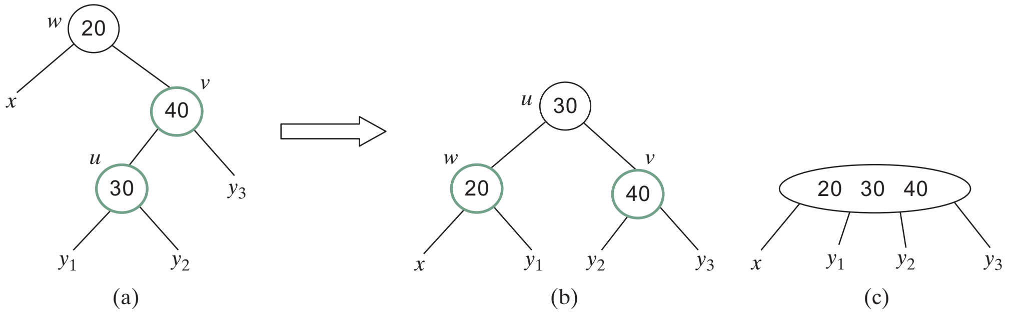


FIGURE 43.10 Case 1.4: $w < u < v$

Case 2: x is red. There are four possible configurations for u, v, w, w, and x, as shown in Figures 43.11(a), 43.11(b), 43.11(c), and 43.11(d). All of these configurations correspond to an overflow situation in the corresponding 4-node in a 2-4 tree, as shown in Figure 43.12(a). A splitting operation is performed to fix the overflow problem in a 2-4 tree, as shown in Figure 43.12(b). We perform an equivalent recoloring operation to fix the problem in a red-black tree. Color w and u red and color two children of w black. Assume u is a left child of v, as shown

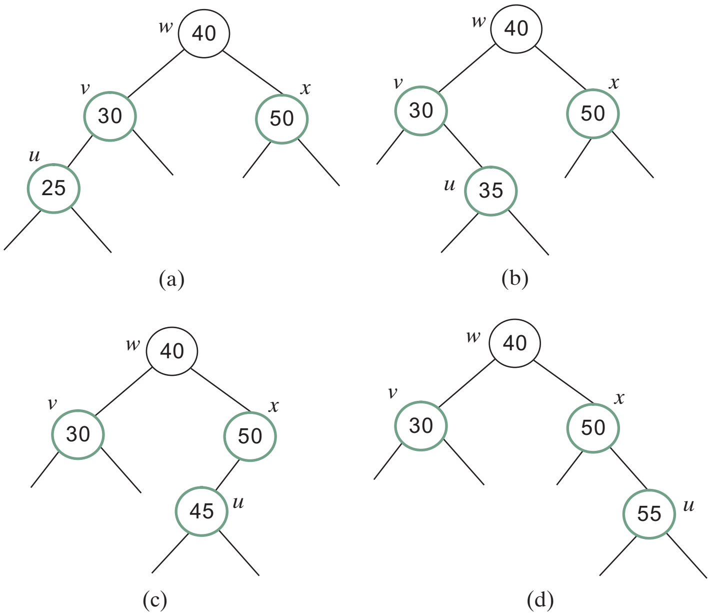


FIGURE 43.11 Case 2 has four possible configurations.

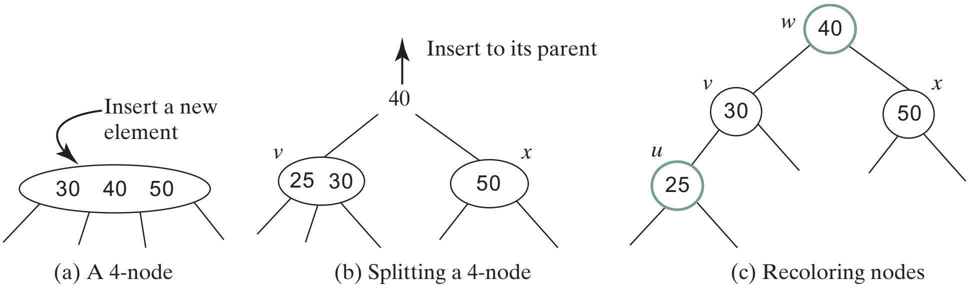


FIGURE 43.12 Splitting a 4-node corresponds to recoloring the nodes in the red-black tree.

in Figure 43.11(a). After recoloring, the nodes are shown in Figure 43.12(c). Now *w* is red, if *w*'s parent is black, the double-red violation is fixed. Otherwise, a new double-red violation occurs at node *w*. We need to continue the same process to eliminate the double-red violation at *w*, recursively.

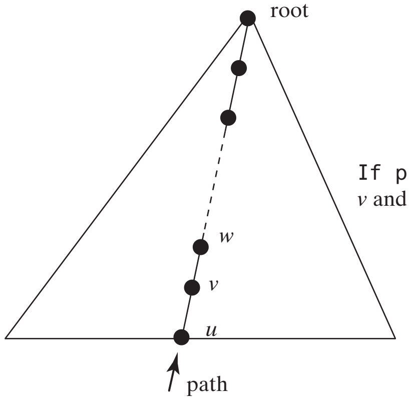
A more detailed algorithm for inserting an element is described in Listing 43.1.

LISTING 43.1 Inserting an Element to a Red-Black Tree

```
public boolean insert(E e) {
  boolean successful = super.insert(e);
  if (!successful)
    return false; // e is already in the tree
  else {
    ensureRBTree(e);
  }

  return true; // e is inserted
}

/** Ensure that the tree is a red-black tree */
private void ensureRBTree(E e) {
  Get the path that leads to element e from the root.
15     int i = path.size() – 1; // Index to the current node in the path
16     Get u, v from the path. u is the node that contains e and v
17       is the parent of u.
18     Color u red;
19
20     if (u == root) // If e is inserted as the root, set root black
21       u.setBlack();
22     else if (v.isRed())
23       fixDoubleRed(u, v, path, i); // Fix double-red violation at u
24   }
25
26   /** Fix double-red violation at node u */
27   private void fixDoubleRed(RBTreeNode<E> u, RBTreeNode<E> v,
28       ArrayList<TreeNode<E>> path, int i) {
29     Get w from the path. w is the grandparent of u.
30
31     // Get v's sibling named x
32     RBTreeNode<E> x = (w.left == v) ?
33       (RBTreeNode<E>)(w.right) : (RBTreeNode<E>)(w.left);
34
35     if (x == null || x.isBlack()) {
36       // Case 1: v's sibling x is black
37       if (w.left == v && v.left == u) {
38         // Case 1.1: u < v < w, Restructure and recolor nodes
39       }
40       else if (w.left == v && v.right == u) {
41         // Case 1.2: v < u < w, Restructure and recolor nodes
42       }
43       else if (w.right == v && v.right == u) {
44         // Case 1.3: w < v < u, Restructure and recolor nodes
45       }
46       else {
47         // Case 1.4: w < u < v, Restructure and recolor nodes
48       }
49     }
50     else { // Case 2: v's sibling x is red
51       Color w and u red
52       Color two children of w black.
53
54       if (w is root) {
55         Set w black;
56       }
57       else if (the parent of w is red) {
58         // Propagate along the path to fix new double-red violation
59         u = w;
60         v = parent of w;
61         fixDoubleRed(u, v, path, i – 2); // i – 2 propagates upward
62       }
63     }
64   }
```

The `insert(E e)` method (lines 1–10) invokes the `insert` method in the `BST` class to create a new leaf node for the element (line 2). If the element is already in the tree, return false (line 4). Otherwise, invoke `ensureRBTree(e)` (line 6) to ensure that the tree satisfies the color and black depth property of the red-black tree.

The `ensureRBTree(E e)` method (lines 13–24) obtains the path that leads to `e` from the root (line 14), as shown in Figure 43.13. This path plays an important role to implement the algorithm. From this path, you get nodes `u` and `v` (lines 16–17). If `u` is the root, color `u` black (lines 20–21). If `v` is red, a double-red violation occurs at node `u`. Invoke `fixDoubleRed` to fix the problem.



`If path.get(i)` is *u*, `path.get(i – 1)` is *v* and `path.get(i – 2)` is *w*.

FIGURE 43.13 The path consists of the nodes from *u* to the root.

The `fixDoubleRed` method (lines 27–63) fixes the double-red violation. It first obtains `w` (the parent of `v`) from the path (line 29) and `x` (the sibling of `v`) (lines 32–33). If `x` is empty or a black node, restructure and recolor three nodes `u`, `v`, and `w` to eliminate the problem (lines 35–49). If `x` is a red node, recolor the nodes `u`, `v`, `w`, and `x` (lines 51–52). If `w` is the root, color `w` black (lines 54–56). If the parent of `w` is red, the double-red violation reappears at `w`. Invoke `fixDoubleRed` with new `u` and `v` to fix the problem (line 61). Note that now `i – 2` points to the new `u` in the path. This adjustment is necessary to locate the new nodes `w` and parent of `w` along the path.

Figure 43.14 shows the steps of inserting `34`, `3`, `50`, `20`, `15`, `16`, `25`, and `27` into an empty red-black tree. When inserting `20` into the tree in (d), Case 2 applies to recolor `3` and `50` to black. When inserting `15` into the tree in (g), Case 1.4 applies to restructure and recolor nodes `15`, `20`, and `3`. When inserting `16` into the tree in (i), Case 2 applies to recolor nodes `3` and `20` to black and nodes `15` and `16` to red. When inserting `27` into the tree in (l), Case 2 applies to recolor nodes `16` and `25` to black and nodes `20` and `27` to red. Now a new double-red problem occurs at node `20`. Apply Case 1.2 to restructure and recolor nodes. The new tree is shown in (n).

43.5 Overriding the `delete` Method

This section discusses how to delete an element to red-black tree.

Key Point

To delete an element from a red-black tree, first search the element in the tree to locate the node that contains the element. If the element is not in the tree, the method returns false. Let *u* be the node that contains the element. If *u* is an internal node with both left and right children, find the rightmost node in the left subtree of *u*. Replace the element in *u* with the element in the rightmost node. Now we will only consider deleting external nodes.

Let *u* be an external node to be deleted. Since *u* is an external node, it has at most one child, denoted by *childOfu*. *childOfu* may be `null`. Let *parentOfu* denote the parent of *u*, as shown in Figure 43.15(a). Delete *u* by connecting *childOfu* with *parentOfu*, as shown in Figure 43.15(b).

Consider the following case:

- If *u* is red, we are done.
- If *u* is black and *childOfu* is red, color *childOfu* black to maintain the black height for *childOfu*.
- Otherwise, assign *childOfu* a fictitious *double black*, as shown in Figure 43.16(a). We call this a *double-black problem*, which indicates that the black depth is short by `1`, caused by deleting a black node `u`.

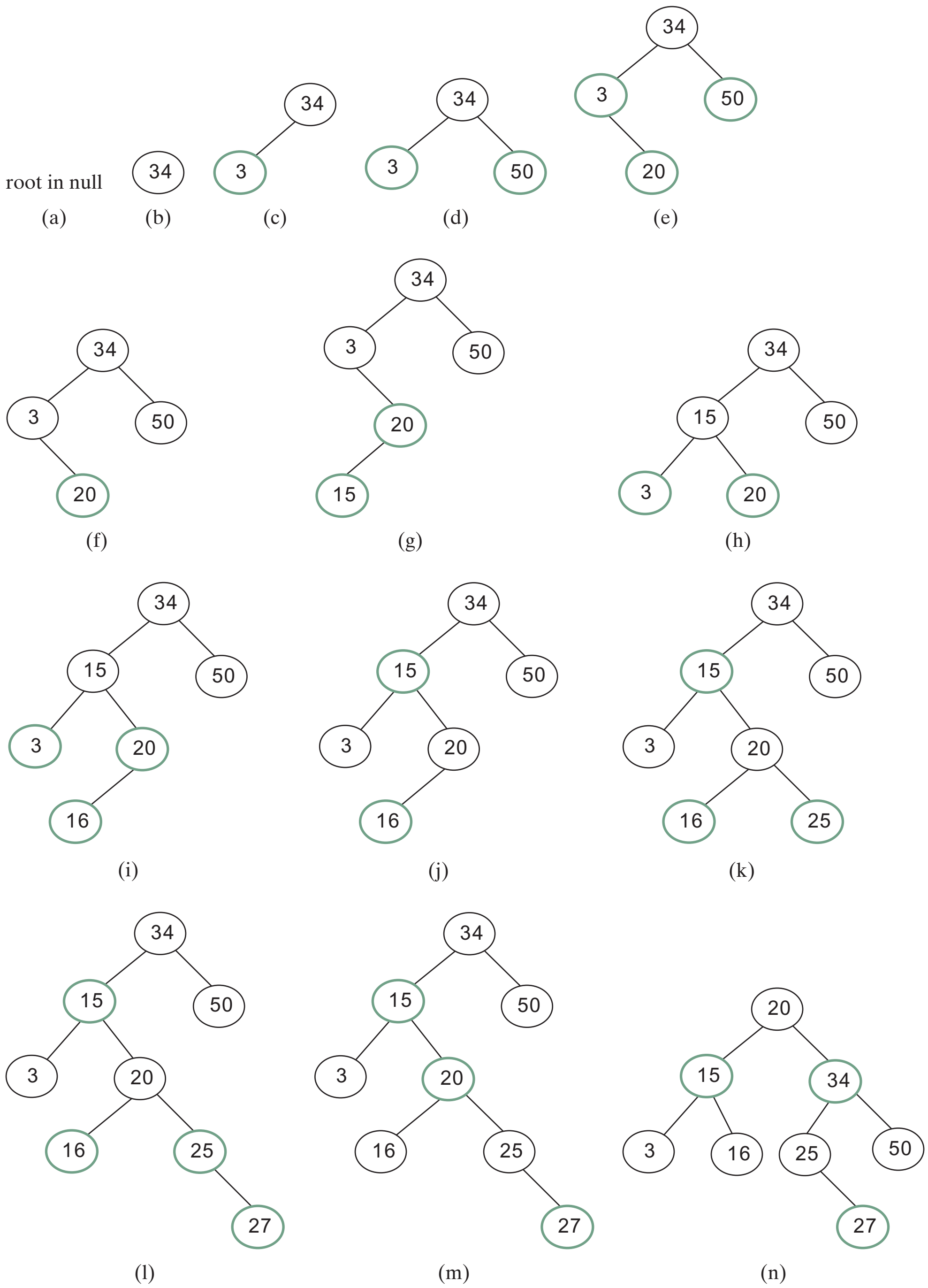
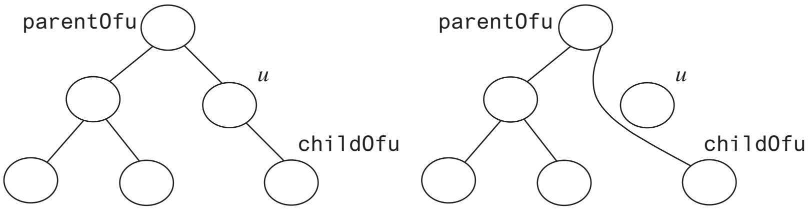
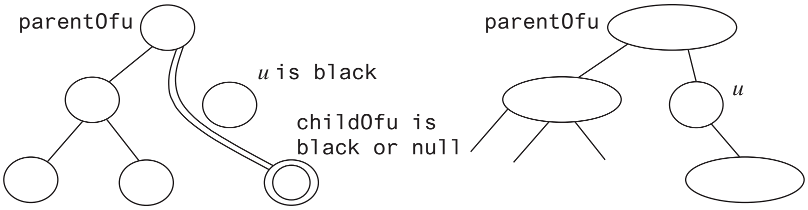


Figure 43.14 Inserting into a red-black tree: (a) initial empty tree; (b) inserting **34**; (c) inserting **3**; (d) inserting **50**; (e) inserting **20** causes a double red; (f) after recoloring (Case 2); (g) inserting **15** causes a double red; (h) after restructuring and recoloring (Case 1.4); (i) inserting **16** causes a double red; (j) after recoloring (Case 2); (k) inserting **25**; (l) inserting **27** causes a double red at **27**; (m) a double red at 20 reappears after recoloring (Case 2); and (n) after restructuring and recoloring (Case 1.2).



(a) Before deleting *u*　　(b) After deleting *u*

FIGURE 43.15 u is an external node and childOfu may be null.

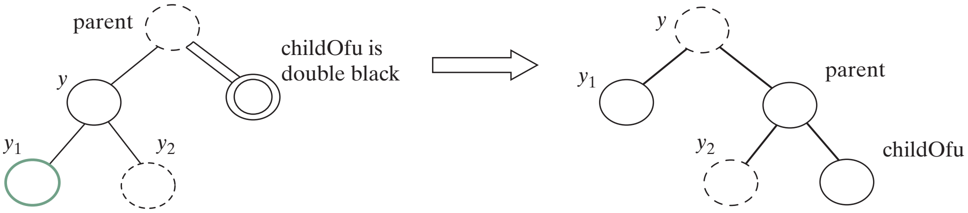


(a)　　(b)

FIGURE 43.16 (a) childOfu is denoted double black. (b) u corresponds to an empty node in a 2-4 tree.

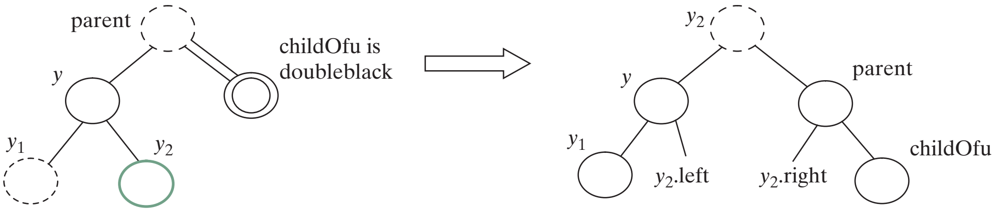
A double black in a red-black tree corresponds to an empty node for u (i.e., underflow situation) in the corresponding 2-4 tree, as shown in Figure 43.16(b). To fix the double-black problem, we will perform equivalent transfer and fusion operations. Consider three cases:

Case 1: The sibling y of childOfu is black and has a red child. This case has four possible configurations, as shown in Figures 43.17(a), 43.18(a), 43.19(a), and 43.20(a). The dashed circle denotes that the node is either red or black. To eliminate the double-black problem, restructure and recolor the nodes, as shown in Figures 43.17(b), 43.18(b), 43.19(b), and 43.20(b).



(a)　　(b)

FIGURE 43.17 Case 1.1: The sibling y of childOfu is black and y1 is red.



(a)　　(b)

FIGURE 43.18 Case 1.2: The sibling y of childOfu is black and y2 is red.

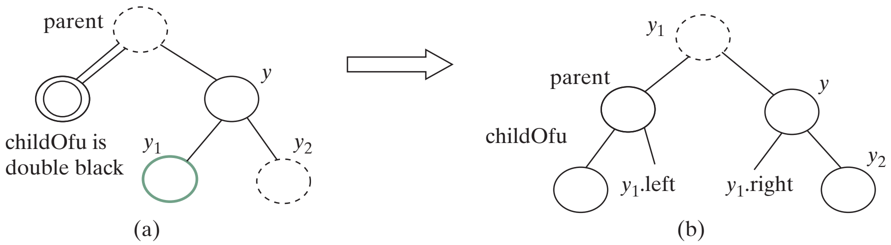


Figure 43.19 Case 1.3: The sibling `y` of `childOfu` is black and `y1` is red.

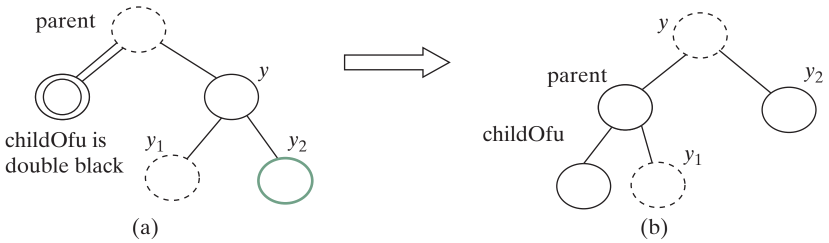


Figure 43.20 Case 1.4: the sibling `y` of `childOfu` is black and `y2` is red.

Note

Case 1 corresponds to a *transfer* operation in the 2-4 tree. For example, the corresponding 2-4 tree for Figure 43.17(a) is shown in Figure 43.21(a), and it is transformed into Figure 43.21(b) through a transfer operation.

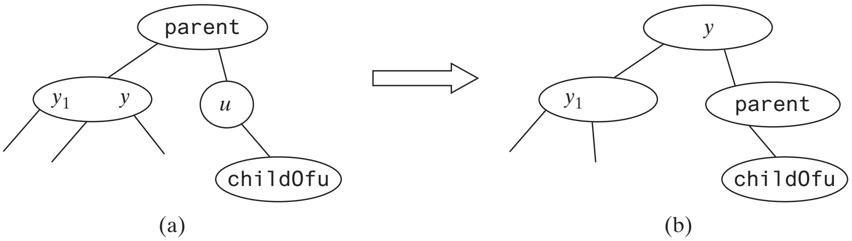


Figure 43.21 Case 1 corresponds to a transfer operation in the corresponding 2-4 tree.

Case 2: The sibling `y` of *childOfu* is black and its children are black or `null`. In this case, change `y`'s color to red. If `parent` is red, change it to black, and we are done, as shown in Figure 43.22. If `parent` is black, we denote `parent` double black, as shown in Figure 43.23. The double-black problem *propagates* to the parent node.

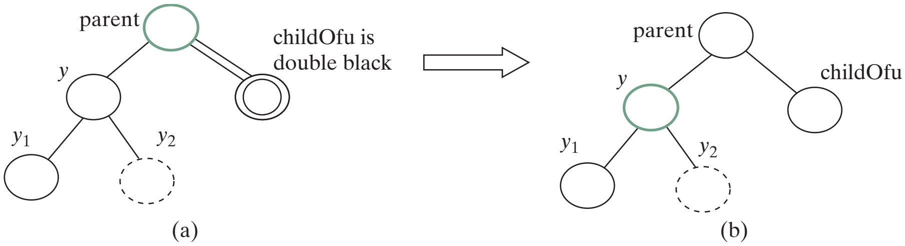


Figure 43.22 Case 2: Recoloring eliminates the double-black problem if `parent` is red.

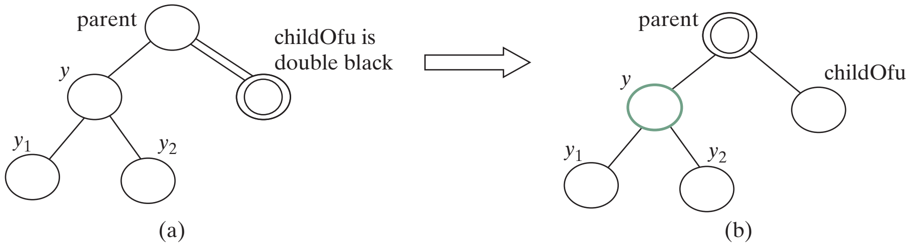


FIGURE 43.23 Case 2: Recoloring propagates the double-black problem if parent is black.

Note
Figures 43.22 and 43.23 show that childOfu is a right child of parent. If childOfu is a left child of parent, recoloring is performed identically.

Note
Case 2 corresponds to a *fusion* operation in the 2-4 tree. For example, the corresponding 2-4 tree for Figure 43.22(a) is shown in Figure 43.24(a), and it is transformed into Figure 43.24(b) through a fusion operation.

Case 3: The sibling y of *childOfu* is red. In this case, perform an *adjustment* operation. If y is a left child of parent, let y1 and y2 be the left and right children of y, as shown in Figure 43.25. If y is a right children of parent, let y1 and y2 be the left and right child of y, as shown in Figure 43.26. In both cases, color y black and parent red. childOfu is still a fictitious double-black node. After the adjustment, the sibling of childOfu is now black, and either Case 1 or Case 2 applies. If Case 1 applies, a one-time restructuring and recoloring operation eliminates the double-black problem. If Case 2 applies, the double-black problem cannot reappear, since parent is now red. Therefore, one-time application of Case 1 or Case 2 will complete Case 3.

Note
Case 3 results from the fact that a 3-node may be transformed in two ways to a red-black tree, as shown in Figure 43.27.

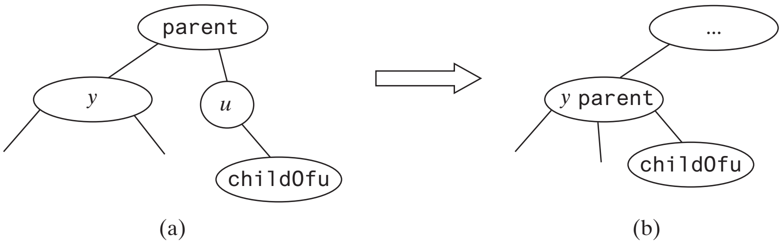


FIGURE 43.24 Case 2 corresponds to a fusion operation in the corresponding 2-4 tree.

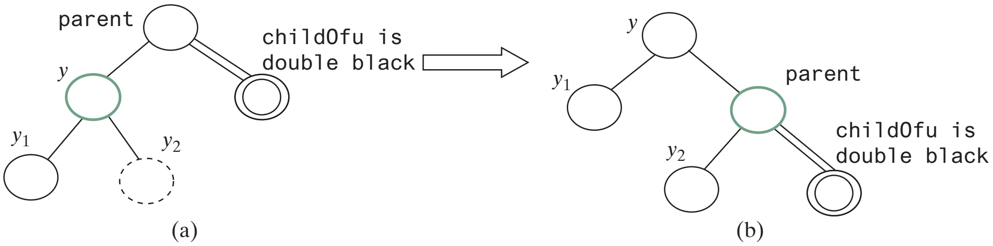


FIGURE 43.25 Case 3.1: y is a left red child of parent.

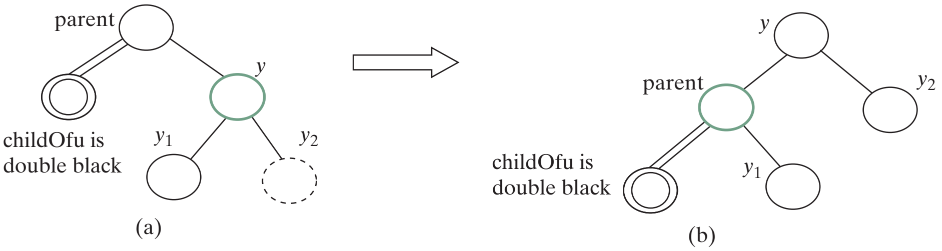


FIGURE 43.26 Case 3.2: `y` is a right red child of `parent`.

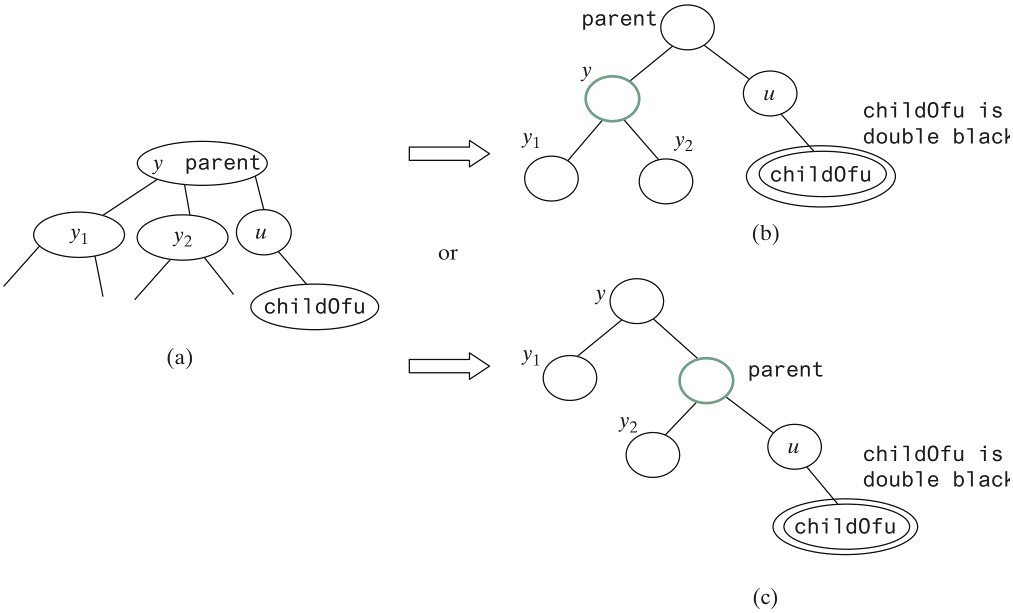


FIGURE 43.27 A 3-node may be transformed in two ways to red-black tree nodes.

Based on the foregoing discussion, Listing 43.2 presents a more detailed algorithm for deleting an element.

LISTING 43.2 Deleting an Element from a Red-Black Tree

```
public boolean delete(E e) {
  Locate the node to be deleted
  if (the node is not found)
    return false;

  if (the node is an internal node) {
    Find the rightmost node in the subtree of the node;
    Replace the element in the node with the one in rightmost;
    The rightmost node is the node to be deleted now;
  }

  Obtain the path from the root to the node to be deleted;

  // Delete the last node in the path and propagate if needed
  deleteLastNodeInPath(path);

```

```
    size--; // After one element deleted
    return true; // Element deleted
  }

  /** Delete the last node from the path. */
  public void deleteLastNodeInPath(ArrayList<TreeNode<e>> path) {
    Get the last node u in the path;
    Get parentOfu and grandparentOfu in the path;
    Get childOfu from u;
    Delete node u. Connect childOfu with parentOfu

    // Recolor the nodes and fix double black if needed
    if (childOfu == root || u.isRed())
      return; // Done if childOfu is root or if u is red
    else if (childOfu != null && childOfu.isRed())
      childOfu.setBlack(); // Set it black, done
    else // u is black, childOfu is null or black
      // Fix double black on parentOfu
      fixDoubleBlack(grandparentOfu, parentOfu, childOfu, path, i);
    }

    /** Fix the double black problem at node parent */
    private void fixDoubleBlack(
        RBTreeNode<E> grandparent, RBTreeNode<E> parent,
        RBTreeNode<E> db, ArrayList<TreeNode<E>> path, int i) {
      Obtain y, y1, and y2

      if (y.isBlack() && y1 != null && y1.isRed()) {
        if (parent.right == db) {
          // Case 1.1: y is a left black sibling and y1 is red
          Restructure and recolor parent, y, and y1 to fix the problem;
        }
        else {
          // Case 1.3: y is a right black sibling and y1 is red
          Restructure and recolor parent, y1, and y to fix the problem;
        }
      }
      else if (y.isBlack() && y2 != null && y2.isRed()) {
        if (parent.right == db) {
          // Case 1.2: y is a left black sibling and y2 is red
          Restructure and recolor parent, y2, and y to fix the problem;
        }
        else {
          // Case 1.4: y is a right black sibling and y2 is red
          Restructure and recolor parent, y, and y2 to fix the problem;
        }
      }
      else if (y.isBlack()) {
        // Case 2: y is black and y's children are black or null
        Recolor y to red;

        if (parent.isRed())
          parent.setBlack(); // Done
        else if (parent != root) {
          // Propagate double black to the parent node
          // Fix new appearance of double black recursively
          db = parent;
          parent = grandparent;
          grandparent =
            (i >= 3) ? (RBTreeNode<E>)(path.get(i - 3)) : null;
```

```
77          fixDoubleBlack(grandparent, parent, db, path, i - 1);
78        }
79      }
80      else if (y.isRed()) {
81        if (parent.right == db) {
82          // Case 3.1: y is a left red child of parent
83          parent.left = y2;
84          y.right = parent;
85        }
86        else {
87          // Case 3.2: y is a right red child of parent
88          parent.right = y.left;
89          y.left = parent;
90        }
91
92        parent.setRed(); // Color parent red
93        y.setBlack(); // Color y black
94        connectNewParent(grandparent, parent, y); // y is new parent
95        fixDoubleBlack(y, parent, db, path, i - 1);
96      }
97    }
```

The `delete(E e)` method (lines 1–19) locates the node that contains `e` (line 2). If the node does not exist, return `false` (lines 3–4). If the node is an internal node, find the right most node in its left subtree and replace the element in the node with the element in the right most node (lines 6–9). Now the node to be deleted is an external node. Obtain the path from the root to the node (line 12). Invoke `deleteLastNodeInPath(path)` to delete the last node in the path and ensure that the tree is still a red-black tree (line 15).

The `deleteLastNodeInPath` method (lines 22–36) obtains the last node `u`, `parentOfu`, `grandparendOfu`, and `childOfu` (lines 23–26). If `childOfu` is the root or `u` is red, the tree is fine (lines 29–30). If `childOfu` is red, color it black (lines 31–32). We are done. Otherwise, `u` is black and `childOfu` is `null` or black. Invoke `fixDoubleBlack` to eliminate the double-black problem (line 35).

The `fixDoubleBlack` method (lines 39–97) eliminates the double-black problem. Obtain `y`, `y1`, and `y2` (line 42). `y` is the sibling of the double-black node. `y1` and `y2` are the left and right children of `y`. Consider three cases:

1. If `y` is black and one of its children is red, the double-black problem can be fixed by one-time restructuring and recoloring in Case 1 (lines 44–63).

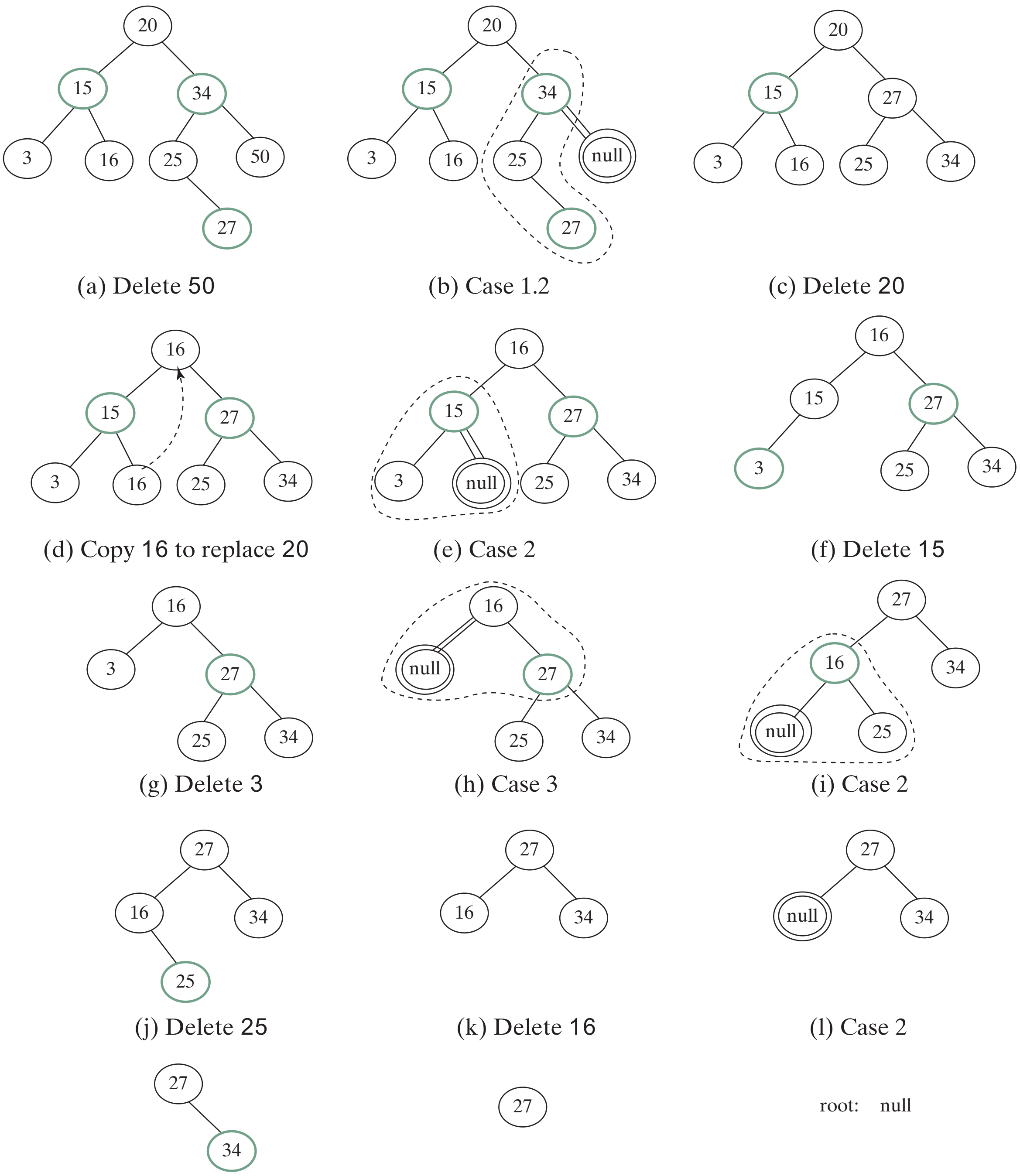
2. If `y` is black and its children are `null` or black, change `y` to red. If `parent` of `y` is black, denote `parent` to be the new double-black node and invoke `fixDoubleBlack` recursively (line 77).

3. If `y` is red, adjust the nodes to make `parent` a child of `y` (lines 84, 89) and color `parent` red and `y` black (lines 92–93). Make `y` the new parent (line 94). Recursively invoke `fixDoubleBlack` on the same double-black node with a different color for `parent` (line 95).

Figure 43.28 shows the steps of deleting elements. To delete `50` from the tree in Figure 43.28(a), apply Case 1.2, as shown in Figure 43.28(b). After restructuring and recoloring, the new tree is as shown in Figure 43.28(c).

When deleting `20` in Figure 43.28(c), 20 is an internal node, and it is replaced by `16`, as shown in Figure 43.28(d). Now Case 2 applies to deleting the rightmost node, as shown in Figure 43.28(e). Recolor the nodes results in a new tree, as shown in Figure 43.28(f).

When deleting `15`, connect node 3 with node 20 and color node 3 black, as shown in Figure 43.28(g). We are done.



(a) Delete 50 (b) Case 1.2 (c) Delete 20

(d) Copy 16 to replace 20 (e) Case 2 (f) Delete 15

(g) Delete 3 (h) Case 3 (i) Case 2

(j) Delete 25 (k) Delete 16 (l) Case 2

(m) Delete 34 (n) Delete 27 (o) Empty tree

FIGURE 43.28 Delete elements from a red-black tree.

After deleting **25**, the new tree is as shown in Figure 43.28(j). Now delete **16**. Apply Case 2, as shown in Figure 43.28(k). The new tree is shown in Figure 43.28(l).

After deleting **34**, the new tree is as shown in Figure 43.28(m).

After deleting **27**, the new tree is as shown in Figure 43.28(n).

Check Point

43.5.1 What are the data fields in RBTreeNode?

43.5.2 How do you insert an element into a red-black tree and how do you fix the double-red violation?

43.5.3 How do you delete an element from a red-black tree and how do you fix the double-black problem?

43.5.4 Show the change of the tree when inserting 1, 2, 3, 4, 10, 9, 7, 5, 8, and 6 into it, in this order.

43.5.5 For the tree built in the preceding question, show the change of the tree after deleting 1, 2, 3, 4, 10, 9, 7, 5, 8, and 6 from it in this order.

43.6 Implementing RBTree Class

Key Point

This section implements the RBTree *class.*

Listing 43.3 gives a complete implementation for the RBTree class.

Listing 43.3 RBTree.java

```java
import java.util.ArrayList;

public class RBTree<E extends Comparable<E>> extends BST<E> {
  /** Create a default RB tree */
  public RBTree() {
  }

  /** Create an RB tree from an array of elements */
  public RBTree(E[] elements) {
    super(elements);
  }

  @Override /** Override createNewNode to create an RBTreeNode */
  protected RBTreeNode<E> createNewNode(E e) {
    return new RBTreeNode<E>(e);
  }

  @Override /** Override the insert method to
    balance the tree if necessary */
  public boolean insert(E e) {
    boolean successful = super.insert(e);
    if (!successful)
      return false; // e is already in the tree
    else {
      ensureRBTree(e);
    }

    return true; // e is inserted
  }

  /** Ensure that the tree is a red-black tree */
  private void ensureRBTree(E e) {
    // Get the path that leads to element e from the root
    ArrayList<TreeNode<E>> path = path(e);

    int i = path.size() - 1; // Index to the current node in the path

    // u is the last node in the path. u contains element e
    RBTreeNode<E> u = (RBTreeNode<E>)(path.get(i));

    // v is the parent of of u, if exists
    RBTreeNode<E> v = (u == root) ? null :
      (RBTreeNode<E>)(path.get(i - 1));

    u.setRed(); // It is OK to set u red

    if (u == root) // If e is inserted as the root, set root black
      u.setBlack();
    else if (v.isRed())
      fixDoubleRed(u, v, path, i); // Fix double-red violation at u
  }

  /** Fix double-red violation at node u */
  private void fixDoubleRed(RBTreeNode<E> u, RBTreeNode<E> v,
      ArrayList<TreeNode<E>> path, int i) {
    // w is the grandparent of u
    RBTreeNode<E> w = (RBTreeNode<E>)(path.get(i - 2));
    RBTreeNode<E> parentOfw = (w == root) ? null :
      (RBTreeNode<E>)path.get(i – 3);

    // Get v's sibling named x
    RBTreeNode<E> x = (w.left == v) ?
      (RBTreeNode<E>)(w.right) : (RBTreeNode<E>)(w.left);

    if (x == null || x.isBlack()) {
      // Case 1: v's sibling x is black
      if (w.left == v && v.left == u) {
        // Case 1.1: u < v < w, Restructure and recolor nodes
        restructureRecolor(u, v, w, w, parentOfw);

        w.left = v.right; // v.right is y3 in Figure 43.6
        v.right = w;
      }
      else if (w.left == v && v.right == u) {
        // Case 1.2: v < u < w, Restructure and recolor nodes
        restructureRecolor(v, u, w, w, parentOfw);
        v.right = u.left;
        w.left = u.right;
        u.left = v;
        u.right = w;
      }
      else if (w.right == v && v.right == u) {
        // Case 1.3: w < v < u, Restructure and recolor nodes
        restructureRecolor(w, v, u, w, parentOfw);
        w.right = v.left;
        v.left = w;
      }
      else {
        // Case 1.4: w < u < v, Restructure and recolor nodes
        restructureRecolor(w, u, v, w, parentOfw);
        w.right = u.left;
        v.left = u.right;
        u.left = w;
        u.right = v;
      }
    }
    else { // Case 2: v's sibling x is red
      // Recolor nodes
      w.setRed();
      u.setRed();
      ((RBTreeNode<E>)(w.left)).setBlack();
      ((RBTreeNode<E>)(w.right)).setBlack();

      if (w == root) {
        w.setBlack();
      }
      else if (((RBTreeNode<E>)parentOfw).isRed()) {
        // Propagate along the path to fix new double-red violation
        u = w;
        v = (RBTreeNode<E>)parentOfw;
        fixDoubleRed(u, v, path, i - 2); // i – 2 propagates upward
      }
    }
  }

  /** Connect b with parentOfw and recolor a, b, c for a < b < c */
  private void restructureRecolor(RBTreeNode<E> a, RBTreeNode<E> b,
      RBTreeNode<E> c, RBTreeNode<E> w, RBTreeNode<E> parentOfw) {
    if (parentOfw == null)
      root = b;
    else if (parentOfw.left == w)
      parentOfw.left = b;
    else
      parentOfw.right = b;

    b.setBlack(); // b becomes the root in the subtree
    a.setRed(); // a becomes the left child of b
    c.setRed(); // c becomes the right child of b
  }

  @Override /** Delete an element from the RBTree.
    * Return true if the element is deleted successfully
    * Return false if the element is not in the tree */
  public boolean delete(E e) {
    // Locate the node to be deleted
    TreeNode<E> current = root;
    while (current != null) {
      if (e.compareTo(current.element) < 0) {
        current = current.left;
      }
      else if (e.compareTo(current.element) > 0) {
        current = current.right;
      }
      else
        break; // Element is in the tree pointed by current
    }

    if (current == null)
      return false; // Element is not in the tree

    java.util.ArrayList<TreeNode<E>> path;

    // current node is an internal node
    if (current.left != null && current.right != null) {
      // Locate the rightmost node in the left subtree of current
      TreeNode<E> rightMost = current.left;
      while (rightMost.right != null) {
        rightMost = rightMost.right; // Keep going to the right
      }

      path = path(rightMost.element); // Get path before replacement

      // Replace the element in current by the element in rightMost
      current.element = rightMost.element;
    }
    else
      path = path(e); // Get path to current node

    // Delete the last node in the path and propagate if needed
    deleteLastNodeInPath(path);

    size--; // After one element deleted
    return true; // Element deleted
  }

  /** Delete the last node from the path. */
  public void deleteLastNodeInPath(ArrayList<TreeNode<E>> path) {
    int i = path.size() - 1; // Index to the node in the path
    // u is the last node in the path
    RBTreeNode<E> u = (RBTreeNode<E>)(path.get(i));
    RBTreeNode<E> parentOfu = (u == root) ? null :
      (RBTreeNode<E>)(path.get(i - 1));
    RBTreeNode<E> grandparentOfu = (parentOfu == null ||
      parentOfu == root) ? null :
      (RBTreeNode<E>)(path.get(i - 2));
    RBTreeNode<E> childOfu = (u.left == null) ?
      (RBTreeNode<E>)(u.right) : (RBTreeNode<E>)(u.left);

    // Delete node u. Connect childOfu with parentOfu
    connectNewParent(parentOfu, u, childOfu);

    // Recolor the nodes and fix double black if needed
    if (childOfu == root || u.isRed())
      return; // Done if childOfu is root or if u is red
    else if (childOfu != null && childOfu.isRed())
      childOfu.setBlack(); // Set it black, done
    else // u is black, childOfu is null or black
      // Fix double black on parentOfu
      fixDoubleBlack(grandparentOfu, parentOfu, childOfu, path, i);
    }

  /** Fix the double-black problem at node parent */
  private void fixDoubleBlack(
      RBTreeNode<E> grandparent, RBTreeNode<E> parent,
      RBTreeNode<E> db, ArrayList<TreeNode<E>> path, int i) {
    // Obtain y, y1, and y2
    RBTreeNode<E> y = (parent.right == db) ?
      (RBTreeNode<E>)(parent.left) : (RBTreeNode<E>)(parent.right);
    RBTreeNode<E> y1 = (RBTreeNode<E>)(y.left);
    RBTreeNode<E> y2 = (RBTreeNode<E>)(y.right);

    if (y.isBlack() && y1 != null && y1.isRed()) {
      if (parent.right == db) {
        // Case 1.1: y is a left black sibling and y1 is red
        connectNewParent(grandparent, parent, y);
        recolor(parent, y, y1); // Adjust colors

        // Adjust child links
        parent.left = y.right;
```

```
          y.right = parent;
        }
        else {
          // Case 1.3: y is a right black sibling and y1 is red
          connectNewParent(grandparent, parent, y1);
          recolor(parent, y1, y); // Adjust colors

          // Adjust child links
          parent.right = y1.left;
          y.left = y1.right;
          y1.left = parent;
          y1.right = y;
        }
      }
      else if (y.isBlack() && y2 != null && y2.isRed()) {
        if (parent.right == db) {
          // Case 1.2: y is a left black sibling and y2 is red
          connectNewParent(grandparent, parent, y2);
          recolor(parent, y2, y); // Adjust colors

          // Adjust child links
          y.right = y2.left;
          parent.left = y2.right;
          y2.left = y;
          y2.right = parent;
         }
        else {
          // Case 1.4: y is a right black sibling and y2 is red
          connectNewParent(grandparent, parent, y);
          recolor(parent, y, y2); // Adjust colors

          // Adjust child links
          y.left = parent;
          parent.right = y1;
        }
      }
      else if (y.isBlack()) {
        // Case 2: y is black and y's children are black or null
        y.setRed(); // Change y to red
        if (parent.isRed())
          parent.setBlack(); // Done
        else if (parent != root) {
          // Propagate double black to the parent node
          // Fix new appearance of double black recursively
          db = parent;
          parent = grandparent;
          grandparent =
            (i >= 3) ? (RBTreeNode<E>)(path.get(i - 3)) : null;
          fixDoubleBlack(grandparent, parent, db, path, i - 1);
        }
      }
      else { // y.isRed()
        if (parent.right == db) {
          // Case 3.1: y is a left red child of parent
          parent.left = y2;
          y.right = parent;
        }
        else {
          // Case 3.2: y is a right red child of parent
          parent.right = y.left;
            y.left = parent;
      }

      parent.setRed(); // Color parent red
      y.setBlack(); // Color y black
      connectNewParent(grandparent, parent, y); // y is new parent
      fixDoubleBlack(y, parent, db, path, i - 1);
    }
  }

  /** Recolor parent, newParent, and c. Case 1 removal */
  private void recolor(RBTreeNode<E> parent,
      RBTreeNode<E> newParent, RBTreeNode<E> c) {
    // Retain the parent's color for newParent
    if (parent.isRed())
      newParent.setRed();
    else
      newParent.setBlack();

    // c and parent become the children of newParent; set them black
    parent.setBlack();
    c.setBlack();
  }

  /** Connect newParent with grandParent */
  private void connectNewParent(RBTreeNode<E> grandparent,
      RBTreeNode<E> parent, RBTreeNode<E> newParent) {
    if (parent == root) {
      root = newParent;
      if (root != null)
        newParent.setBlack();
    }
    else if (grandparent.left == parent)
      grandparent.left = newParent;
    else
      grandparent.right = newParent;
  }

  @Override /** Preorder traversal from a subtree */
  protected void preorder(TreeNode<E> root) {
    if (root == null) return;
    System.out.print(root.element +
      (((RBTreeNode<E>)root).isRed() ? " (red) " : " (black) "));
    preorder(root.left);
    preorder(root.right);
  }

  /** RBTreeNode is TreeNode plus color indicator */
  protected static class RBTreeNode<E extends Comparable<E>> extends
      BST.TreeNode<E> {
    private boolean red = true; // Indicate node color

    public RBTreeNode(E e) {
      super(e);
    }

    public boolean isRed() {
      return red;
    }

    public boolean isBlack() {
      return !red;
```

```
342      }
343
344      public void setBlack() {
345        red = false;
346      }
347
348      public void setRed() {
349        red = true;
350      }
351
352      int blackHeight;
353    }
354  }
```

The `RBTree` class extends `BST`. Like the `BST` class, the `RBTree` class has a no-arg constructor that constructs an empty `RBTree` (lines 5–6) and a constructor that creates an initial `RBTree` from an array of elements (lines 9–11).

The `createNewNode()` method defined in the `BST` class creates a `TreeNode`. This method is overridden to return an `RBTreeNode` (lines 14–16). This method is invoked in the insert method in `BST` to create a node.

The `insert` method in `RBTree` is overridden in lines 20–29. The method first invokes the `insert` method in `BST`, then invokes `ensureRBTree(e)` (line 25) to ensure that tree is still a red-black tree after inserting a new element.

The `ensureRBTree(E e)` method first obtains the path of nodes that lead to element `e` from the root (line 34). It obtains `u` and `v` (the parent of `u`) from the path. If `u` is the root, color `u` black (lines 47–48). If `v` is red, invoke `fixDoubleRed` to fix the double red on both `u` and `v` (lines 49–50).

The `fixDoubleRed(u, v, path, i)` method fixes the double-red violation at node `u`. The method first obtains `w` (the grandparent of `u` from the path) (line 57), `parentOfw` if exists (lines 58–59), and `x` (the sibling of `v`) (lines 62–63). If `x` is `null` or black, consider four subcases to fix the double-red violation (lines 67–96). If x is red, color `w` and `u` red and color `w`'s two children black (lines 101–104). If `w` is the root, color `w` black (lines 104–106). Otherwise, propagate along the path to fix the new double-red violation (lines 109–111).

The `delete(E e)` method in `RBTree` is overridden in lines 134–174. The method locates the node that contains `e` (lines 136–146). If the node is null, no element is found (lines 148–149). The method considers two cases:

- If the node is internal, find the rightmost node in its left subtree (lines 156–159). Obtain a path from the root to the rightmost node (line 161), and replace the element in the node with the element in the rightmost node (line 164).
- If the node is external, obtain the path from the root to the node (line 167).

The last node in the path is the node to be deleted. Invoke `deleteLastNodeInPath(path)` to delete it and ensure the tree is a red-black after the node is deleted (line 170).

The `deleteLastNodeInPath(path)` method first obtains `u`, `parentOfu`, `grandparendOfu`, and `childOfu` (lines 180–187). `u` is the last node in the path. Connect `childOfu` as a child of `parentOfu` (line 190). This in effect deletes `u` from the tree. Consider three cases:

- If `childOfu` is the root or `childOfu` is red, we are done (lines 193–194).
- Otherwise, if `childOfu` is red, color it black (lines 195–196).
- Otherwise, invoke `fixDoubleBlack` to fix the double-black problem on `childOfu` (line 199).

The `fixDoubleBlack` method first obtains `y`, `y1`, and `y2` (lines 207–210). `y` is the sibling of the first double-black node, and `y1` and `y2` are the left and right children of `y`. Consider three cases:

- If `y` is black and `y1` or `y2` is red, fix the double-black problem for Case 1 (lines 213–255).
- Otherwise, if `y` is black, fix the double-black problem for Case 2 by recoloring the nodes. If parent is black and not a root, propagate double black to parent and recursively invoke `fixDoubleBlack` (lines 264–268).
- Otherwise, `y` is red. In this case, adjust the nodes to make parent the child of `y` (lines 272–281). Invoke `fixDoubleBlack` with the adjusted nodes (line 286) to fix the double-black problem.

The `preorder(TreeNode<E> root)` method is overridden to display the node colors (lines 319–325).

43.7 Testing the RBTree Class

This section gives a test program that uses the **`RBTree`** *class.*

Key Point

Listing 43.4 gives a test program. The program creates an `RBTree` initialized with an array of integers `34`, `3`, and `50` (lines 4–5), inserts elements in lines 10–22, and deletes elements in lines 25–46.

Listing 43.4 TestRBTree.java

```
 1  public class TestRBTree {
 2    public static void main(String[] args) {
 3      // Create an RB tree
 4      RBTree<Integer> tree =
 5        new RBTree<Integer>(new Integer[]{34, 3, 50});
 6      printTree(tree);
 7
 8      tree.insert(20);
 9      printTree(tree);
10
11      tree.insert(15);
12      printTree(tree);
13
14      tree.insert(16);
15      printTree(tree);
16
17      tree.insert(25);
18      printTree(tree);
19
20      tree.insert(27);
21      printTree(tree);
22
23      tree.delete(50);
24      printTree(tree);
25
26      tree.delete(20);
27      printTree(tree);
28
29      tree.delete(15);
30      printTree(tree);
31
```

```
    tree.delete(3);
    printTree(tree);

    tree.delete(25);
    printTree(tree);

    tree.delete(16);
    printTree(tree);

    tree.delete(34);
    printTree(tree);

    tree.delete(27);
    printTree(tree);
  }

  public static <E extends Comparable<E>>
    void printTree(BST <E> tree) {
    // Traverse tree
    System.out.print("\nInorder (sorted): ");
    tree.inorder();
    System.out.print("\nPostorder: ");
    tree.postorder();
    System.out.print("\nPreorder: ");
    tree.preorder();
    System.out.print("\nThe number of nodes is " + tree.getSize());
    System.out.println();
  }
}
```

```
Inorder (sorted): 3 34 50
Postorder: 3 50 34
Preorder: 34 (black) 3 (red) 50 (red)
The number of nodes is 3

Inorder (sorted): 3 20 34 50
Postorder: 20 3 50 34
Preorder: 34 (black) 3 (black) 20 (red) 50 (black)
The number of nodes is 4

Inorder (sorted): 3 15 20 34 50
Postorder: 3 20 15 50 34
Preorder: 34 (black) 15 (black) 3 (red) 20 (red) 50 (black)
The number of nodes is 5

Inorder (sorted): 3 15 16 20 34 50
Postorder: 3 16 20 15 50 34
Preorder: 34 (black) 15 (red) 3 (black) 20 (black) 16 (red) 50 (black)
The number of nodes is 6

Inorder (sorted): 3 15 16 20 25 34 50
Postorder: 3 16 25 20 15 50 34
Preorder: 34 (black) 15 (red) 3 (black) 20 (black) 16 (red) 25 (red)
  50 (black)
The number of nodes is 7

Inorder (sorted): 3 15 16 20 25 27 34 50
Postorder: 3 16 15 27 25 50 34 20
```

```
Preorder: 20 (black) 15 (red) 3 (black) 16 (black) 34 (red) 25 (black)
  27 (red) 50 (black)
The number of nodes is 8

Inorder (sorted): 3 15 16 20 25 27 34
Postorder: 3 16 15 25 34 27 20
Preorder: 20 (black) 15 (red) 3 (black) 16 (black) 27 (red)
  25 (black) 34 (black)
The number of nodes is 7

Inorder (sorted): 3 15 16 25 27 34
Postorder: 3 15 25 34 27 16
Preorder: 16 (black) 15 (black) 3 (red) 27 (red) 25 (black) 34 (black)
The number of nodes is 6

Inorder (sorted): 3 16 25 27 34
Postorder: 3 25 34 27 16
Preorder: 16 (black) 3 (black) 27 (red) 25 (black) 34 (black)
The number of nodes is 5

Inorder (sorted): 16 25 27 34
Postorder: 25 16 34 27
Preorder: 27 (black) 16 (black) 25 (red) 34 (black)
The number of nodes is 4

Inorder (sorted): 16 27 34
Postorder: 16 34 27
Preorder: 27 (black) 16 (black) 34 (black)
The number of nodes is 3

Inorder (sorted): 27 34
Postorder: 34 27
Preorder: 27 (black) 34 (red)
The number of nodes is 2

Inorder (sorted): 27
Postorder: 27
Preorder: 27 (black)
The number of nodes is 1

Inorder (sorted):
Postorder:
Preorder:
The number of nodes is 0
```

Figure 43.14 shows how the tree evolves as elements are added to it, and Figure 43.28 shows how the tree evolves as elements are deleted from it.

43.8 Performance of the **RBTree** Class

This search, insertion, and deletion operations take O(logn) time in a red-black tree.

Key Point

The search, insertion, and deletion times in a red-black tree depend on the height of the tree. A red-black tree corresponds to a 2–4 tree. When you convert a node in a 2–4 tree to red-black tree nodes, you get one black node and zero, one, or two red nodes as its children, depending on whether the original node is a 2-node, 3-node, or 4-node. So, the height of a red-black tree

Table 43.1 Time Complexities for Methods in `RBTree`, `AVLTree`, and `Tree234`

Methods	*Red-Black Tree*	*AVL Tree*	*2-4 Tree*
`search (e: E)`	*O*(log*n*)	*O*(log*n*)	*O*(log*n*)
`insert (e: E)`	*O*(log*n*)	*O*(log*n*)	*O*(log*n*)
`delete (e: E)`	*O*(log*n*)	*O*(log*n*)	*O*(log*n*)
`getSize()`	*O*(1)	*O*(1)	*O*(1)
`isEmpty()`	*O*(1)	*O*(1)	*O*(1)

is at most as twice that of its corresponding 2–4 tree. Since the height of a 2–4 tree is log *n*, the height of a red-black tree is 2log *n*.

A red-black tree has the same time complexity as an AVL tree, as shown in Table 43.1. In general, a red-black is more efficient than an AVL tree, because a red-black tree requires only one-time restructuring of the nodes for insert and delete operations.

A red-black tree has the same time complexity as a 2–4 tree, as shown in Table 43.1. In general, a red-black is more efficient than a 2–4 tree for two reasons:

1. A red-black tree requires only one-time restructuring of the nodes for insert and delete operations. However, a 2–4 tree may require many splits for an insert operation and fusion for a delete operation.
2. A red-black tree is a binary search tree. A binary tree can be implemented more space efficiently than a 2–4 tree, because a node in a 2–4 tree has at most three elements and four children. Space is wasted for 2-nodes and 3-nodes in a 2–4 tree.

Listing 43.5 gives an empirical test of the performance of AVL trees, 2–4 trees, and red-black trees.

Listing 43.5 TreePerformanceTest.java

```java
public class TreePerformanceTest {
  public static void main(String[] args) {
    final int TEST_SIZE = 500000; // Tree size used in the test

    // Create an AVL tree
    Tree<Integer> tree1 = new AVLTree<Integer>();
    System.out.println("AVL tree time: " +
      getTime(tree1, TEST_SIZE) + " milliseconds");

    // Create a 2-4 tree
    Tree<Integer> tree2 = new Tree24<Integer>();
    System.out.println("2-4 tree time: "
      + getTime(tree2, TEST_SIZE) + " milliseconds");

    // Create a red-black tree
    Tree<Integer> tree3 = new RBTree<Integer>();
    System.out.println("RB tree time: "
      + getTime(tree3, TEST_SIZE) + " milliseconds");
  }

  public static long getTime(Tree<Integer> tree, int testSize) {
    long startTime = System.currentTimeMillis(); // Start time

    // Create a list to store distinct integers
    java.util.List<Integer> list = new java.util.ArrayList<Integer>();
26       for (int i = 0; i < testSize; i++)
27         list.add(i);
28
29       java.util.Collections.shuffle(list); // Shuffle the list
30
31       // Insert elements in the list to the tree
32       for (int i = 0; i < testSize; i++)
33         tree.insert(list.get(i));
34
35       java.util.Collections.shuffle(list); // Shuffle the list
36
37       // Delete elements in the list from the tree
38       for (int i = 0; i < testSize; i++)
39         tree.delete(list.get(i));
40
41       // Return elapse time
42       return System.currentTimeMillis() - startTime;
43     }
44   }
```

```
AVL tree time: 7609 milliseconds
2-4 tree time: 8594 milliseconds
RB tree time: 5515 milliseconds
```

The `getTestTime` method creates a list of distinct integers from `0` to `testSize - 1` (lines 25–27), shuffles the list (line 29), adds the elements from the list to a tree (lines 32–33), shuffles the list again (line 35), removes the elements from the tree (lines 38–39), and finally returns the execution time (line 42).

The program creates an AVL (line 6), a 2-4 tree (line 11), and a red-black tree (line 16). The program obtains the execution time for adding and removing `500000` elements in the three trees.

As you see, the red-black tree performs the best, followed by the AVL tree.

Note

The `java.util.TreeSet` class in the Java API is implemented using a red-black tree. Each entry in the set is stored in the tree. Since the `search`, `insert`, and `delete` methods in a red-black tree take $O(\log n)$ time, the `get`, `add`, `remove`, and `contains` methods in `java.util.TreeSet` take $O(\log n)$ time.

Note

The `java.util.TreeMap` class in the Java API is implemented using a red-black tree. Each entry in the map is stored in the tree. The order of the entries is determined by their keys. Since the `search`, `insert`, and `delete` methods in a red-black tree take $O(\log n)$ time, the `get`, `put`, `remove`, and `containsKey` methods in `java.util.TreeMap` take $O(\log n)$ time.

Key Terms

black depth	43-2
double-black violation	43-11
double-red violation	43-7
external node	43-9
red-black tree	43-2

Chapter Summary

1. A red-black tree is a binary search tree, derived from a *2-4 tree*. A red-black tree corresponds to a 2-4 tree. You can convert a red-black tree to a 2-4 tree or vice versa.

2. In a red-black tree, each node is colored red or black. The root is always black. Two adjacent nodes cannot be both red. All external nodes have the same black depth.

3. Since a red-black tree is a binary search tree, the `RBTree` class extends the `BST` class.

4. Searching an element in a red-black tree is the same as in binary search tree, since a red-black tree is a binary search tree.

5. A new element is always inserted as a leaf node. If the new node is the root, color it black. Otherwise, color it red. If the parent of the new node is red, we have to fix the *double-red violation* by reassigning the color and/or restructuring the tree.

6. If a node to be deleted is internal, find the rightmost node in its left subtree. Replace the element in the node with the element in the rightmost node. Delete the rightmost node.

7. If the external node to be deleted is red, simply reconnect the parent node of the external node with the child node of the external node.

8. If the external node to be deleted is black, you need to consider several cases to ensure that black height for external nodes in the tree is maintained correctly.

9. The height of a red-black tree is $O(\log n)$. So, the time complexities for the `search`, `insert`, and `delete` methods are $O(\log n)$.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

Programming Exercises

*43.1 (*red-black tree to 2-4 tree*) Write a program that converts a red-black tree to a 2-4 tree.

*43.2 (*2-4 tree to red-black tree*) Write a program that converts a red-black tree to a 2-4 tree.

***43.3 (*red-black tree animation*) Write a GUI program that animates the red-black tree `insert`, `delete`, and `search` methods, as shown in Figure 43.6.

**43.4 (*Parent reference for `RBTree`*) Suppose the `TreeNode` class defined in `BST` contains a reference to the node's parent, as shown in Exercise 26.17. Implement the `RBTree` class to support this change. Write a test program that adds numbers `1`, `2`, . . . , `100` to the tree and displays the paths for all leaf nodes.

CHAPTER

44

Testing Using JUnit

Objectives

- To know what JUnit is and how JUnit works (§44.2).
- To create and run a JUnit test class from the command window (§44.2).
- To create and run a JUnit test class from NetBeans (§44.3).
- To create and run a JUnit test class from Eclipse (§44.4).

44.1 Introduction

JUnit is a tool for testing Java programs.

At the very beginning of this book in Section 2.16, we introduced software development process that includes requirements specification, analysis, design, implementation, testing, deployment, and maintenance. Testing is an important part of this process. This chapter introduces how to test Java classes using JUnit.

44.2 JUnit Basics

To test a class, you need to write a test class and run it through JUnit to generate a report for the class.

JUnit is the de facto framework for testing Java programs. JUnit is a third-party open-source library packed in a jar file. The jar file contains a tool called *test runner*, which is used to run test programs. Suppose you have a class named `A`. To test this class, you write a test class named `ATest`. This test class, called a *test runner*, contains the methods you write for testing class `A`. The test runner executes `ATest` to generate a test report, as shown in Figure 44.1.

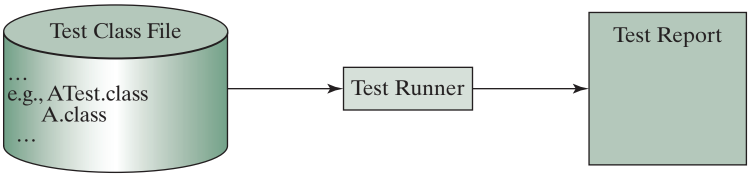


FIGURE 44.1 JUnit test runner executes the test class to generate a test report.

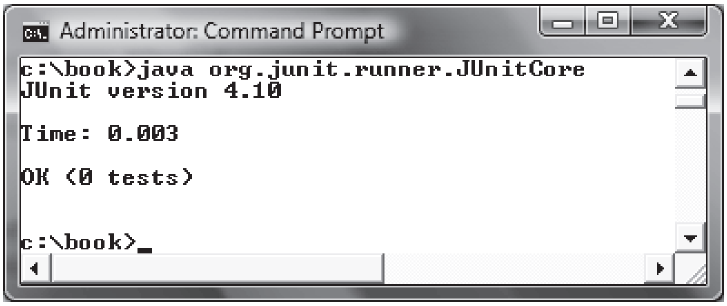
You will see how JUnit works from an example. To create the example, first you need to download JUnit from http://sourceforge.net/projects/junit/files/. At present, the latest version is junit-4.10.jar. Download this file to c:\book\lib and add it to the classpath environment variable as follows:

```
set classpath=.;%classpath%;c:\book\lib\junit-4.10.jar
```

To test if this environment variable is set correctly, open a new command window, and type the following command:

```
java org.junit.runner.JUnitCore
```

You should see the message displayed as shown in Figure 44.2.



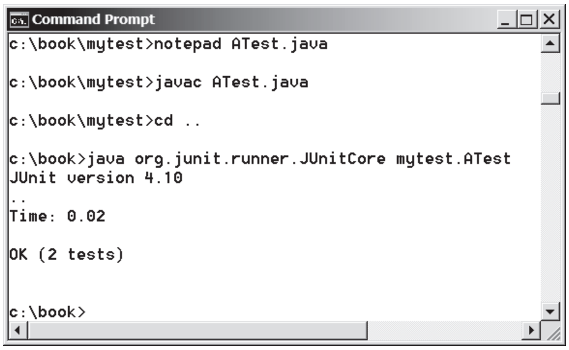
```
c:\book>java org.junit.runner.JUnitCore
JUnit version 4.10

Time: 0.003

OK (0 tests)


c:\book>_
```

FIGURE 44.2 The JUnit test runner displays the JUnit version.

To use JUnit, create a test class. By convention, if the class to be tested is named `A`, the test class should be named `ATest`. A simple template of a test class may look like this:

```java
package mytest;

import org.junit.*;
import static org.junit.Assert.*;

public class ATest {
  @Test
  public void m1() {
    // Write a test method
  }

  @Test
  public void m2() {
    // Write another test method
  }

  @Before
  public void setUp() throws Exception {
    // Common objects used by test methods may be set up here
  }
}
```

This class should be placed in a directory under mytest. Suppose the class is placed under c:\book\mytest. You need to compile it from the mytest directory and run it from c:\book as shown in the following screen shot.



```
c:\book\mytest>notepad ATest.java

c:\book\mytest>javac ATest.java

c:\book\mytest>cd ..

c:\book>java org.junit.runner.JUnitCore mytest.ATest
JUnit version 4.10
..
Time: 0.02

OK (2 tests)


c:\book>
```

Note the command to run the test from the console is:

```
java org.junit.runner.JUnitCore mytest.ATest
```

When this command is executed, `JUnitCore` controls the execution of `ATest`. It first executes the `setUp()` method to set up the common objects used for the test, and then executes test methods `m1` and `m2` in this order. You may define multiple test methods if desirable.

The following methods can be used to implement a test method:

`assertTrue(booleanExpression)`
The method reports success if the booleanExpression evaluates true.

`assertEquals(Object, Object)`
The method reports success if the two objects are the same using the `equals` method.

`assertNull(Object)`
The method reports success if the object reference passed is `null`.

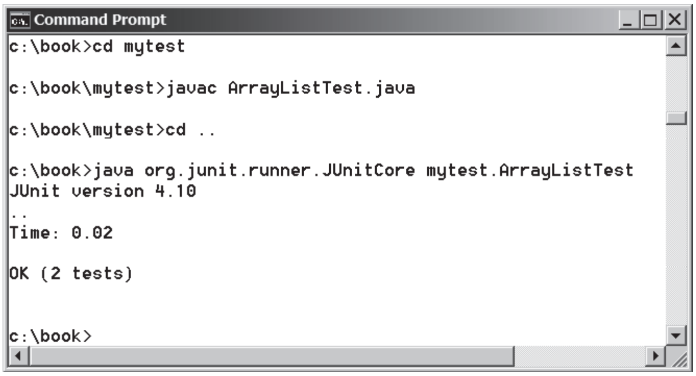
`fail(String)`
The method causes the test to fail and prints out the string.

Listing 44.1 is an example of a test class for testing `java.util.ArrayList`.

Listing 44.1 ArrayListTest.java

```java
package mytest;

import org.junit.*;
import static org.junit.Assert.*;
import java.util.*;

public class ArrayListTest {
  private ArrayList<String> list = new ArrayList<String>();

  @Before
  public void setUp() throws Exception {
  }

  @Test
  public void testInsertion() {
    list.add("Beijing");
    assertEquals("Beijing", list.get(0));
    list.add("Shanghai");
    list.add("Hongkong");
    assertEquals("Hongkong", list.get(list.size() - 1));
  }

  @Test
  public void testDeletion() {
    list.clear();
    assertTrue(list.isEmpty());

    list.add("A");
    list.add("B");
    list.add("C");
    list.remove("B");
    assertEquals(2, list.size());
  }
}
```

A test run of the program is shown in Figure 44.3. Note that you have to first compile ArrayListTest.java. The `ArrayListTest` class is placed in the `mytest` package. So you should place ArrayListTest.java in the directory named `mytest`.



```
c:\book>cd mytest

c:\book\mytest>javac ArrayListTest.java

c:\book\mytest>cd ..

c:\book>java org.junit.runner.JUnitCore mytest.ArrayListTest
JUnit version 4.10
..
Time: 0.02

OK (2 tests)


c:\book>
```

Figure 44.3 The test report is displayed from running `ArrayListTest`.

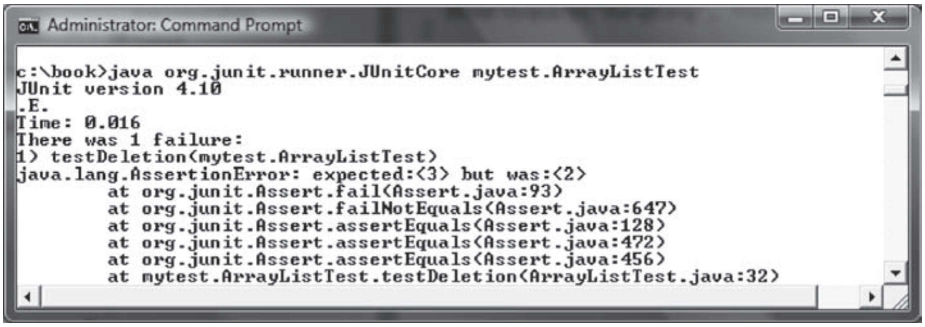
No errors are reported in this JUnit run. If you mistakenly change

```
assertEquals(2, list.size());
```

in line 32 to

```
assertEquals(3, list.size());
```

Run ArrayListTest now. You will see an error reported as shown in Figure 44.4.



```
c:\book>java org.junit.runner.JUnitCore mytest.ArrayListTest
JUnit version 4.10
.E.
Time: 0.016
There was 1 failure:
1) testDeletion(mytest.ArrayListTest)
java.lang.AssertionError: expected:<3> but was:<2>
        at org.junit.Assert.fail(Assert.java:93)
        at org.junit.Assert.failNotEquals(Assert.java:647)
        at org.junit.Assert.assertEquals(Assert.java:128)
        at org.junit.Assert.assertEquals(Assert.java:472)
        at org.junit.Assert.assertEquals(Assert.java:456)
        at mytest.ArrayListTest.testDeletion(ArrayListTest.java:32)
```

Figure 44.4 The test report reports an error.

You can define any number of test methods. In this example, the two test methods testInsertion and testDeletion are defined. JUnit executes testInsertion and testDeletion in this order.

Note
The test class must be placed in a named package such as mytest in this example. The JUnit will not work if the test class is placed a default package.

Listing 44.2 gives a test class for testing the Loan class in Listing 10.2. For convenience, we create Loan.java in the same directory with LoanTest.java. The Loan class is shown in Listing 44.3.

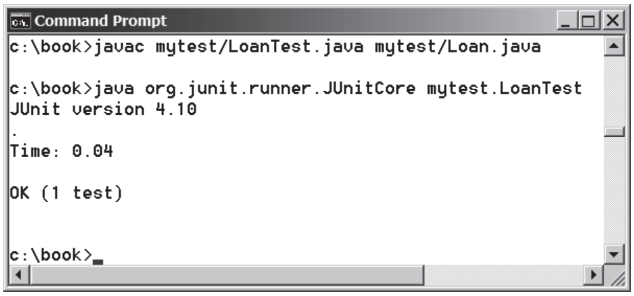
Listing 44.2 LoanTest.java

```
package mytest;

import org.junit.*;
import static org.junit.Assert.*;

public class LoanTest {
  @Before
  public void setUp() throws Exception {
  }

  @Test
  public void testPaymentMethods() {
    double annualInterestRate = 2.5;
    int numberOfYears = 5;
    double loanAmount = 1000;
    Loan loan = new Loan(annualInterestRate, numberOfYears,
      loanAmount);

    assertTrue(loan.getMonthlyPayment() ==
      getMonthlyPayment(annualInterestRate, numberOfYears,
      loanAmount));
    assertTrue(loan.getTotalPayment() ==
      getTotalPayment(annualInterestRate, numberOfYears,
      loanAmount));
  }

  /** Find monthly payment */
  private double getMonthlyPayment(double annualInterestRate,
      int numberOfYears, double loanAmount) {
    double monthlyInterestRate = annualInterestRate / 1200;
    double monthlyPayment = loanAmount * monthlyInterestRate / (1 –
      (1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12)));
    return monthlyPayment;
  }

  /** Find total payment */
  public double getTotalPayment(double annualInterestRate,
      int numberOfYears, double loanAmount) {
    return getMonthlyPayment(annualInterestRate, numberOfYears,
      loanAmount) * numberOfYears * 12;
  }
}
```

Listing 44.3 Loan.java

```
package mytest;

public class Loan {
  private double annualInterestRate;
  private int numberOfYears;
  private double loanAmount;
  private java.util.Date loanDate;

  /** Default constructor */
  public Loan() {
    this(2.5, 1, 1000);
  }

  /** Construct a loan with specified annual interest rate,
      number of years, and loan amount
    */
  public Loan(double annualInterestRate, int numberOfYears,
      double loanAmount) {
    this.annualInterestRate = annualInterestRate;
    this.numberOfYears = numberOfYears;
    this.loanAmount = loanAmount;
    loanDate = new java.util.Date();
  }

  /** Return annualInterestRate */
  public double getAnnualInterestRate() {
    return annualInterestRate;
  }

  /** Set a new annualInterestRate */
  public void setAnnualInterestRate(double annualInterestRate) {
    this.annualInterestRate = annualInterestRate;
  }

  /** Return numberOfYears */
  public int getNumberOfYears() {
    return numberOfYears;
  }

  /** Set a new numberOfYears */
  public void setNumberOfYears(int numberOfYears) {
    this.numberOfYears = numberOfYears;
  }

  /** Return loanAmount */
  public double getLoanAmount() {
    return loanAmount;
  }

  /** Set a newloanAmount */
  public void setLoanAmount(double loanAmount) {
    this.loanAmount = loanAmount;
  }

  /** Find monthly payment */
  public double getMonthlyPayment() {
    double monthlyInterestRate = annualInterestRate / 1200;
    double monthlyPayment = loanAmount * monthlyInterestRate / (1 –
      (1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12)));
    return monthlyPayment;
  }

  /** Find total payment */
  public double getTotalPayment() {
    double totalPayment = getMonthlyPayment() * numberOfYears * 12;
    return totalPayment;
  }

  /** Return loan date */
  public java.util.Date getLoanDate() {
    return loanDate;
  }
}
```

The `testPaymentMethods()` in `LoanTest` creates an instance of `Loan` (line 16–17) and tests whether `loan.getMonthlyPayment()` returns the same value as `getMonthlyPayment(annualInterestRate, numberOfYears, loanAmount)`. The latter method is defined in the `LoanTest` class (lines 28–34).



```
c:\book>javac mytest/LoanTest.java mytest/Loan.java

c:\book>java org.junit.runner.JUnitCore mytest.LoanTest
JUnit version 4.10
.
Time: 0.04

OK (1 test)


c:\book>_
```

FIGURE 44.5 The JUnit test runner executes `LoanTest` and reports no errors.

The `testPaymentMethods()` also tests whether `loan.getTotalPayment()` returns the same value as `getTotalPayment(annualInterestRate, numberOfYears, loanAmount)`. The latter method is defined in the `LoanTest` class (lines 37–41).

A sample run of the program is shown in Figure 44.5.

Check Point

44.2.1 What is JUnit?

44.2.2 What is a JUnit test runner?

44.2.3 What is a test class? How do you create a test class?

44.2.4 How do you use the `assertTrue` method?

44.2.5 How do you use the `assertEquals` method?

44.3 Using JUnit from NetBeans

Key Point

JUnit is intergrated with NetBeans. Using NetBeans, the test program can be automatically generated and the test process can be automated.

An IDE such as NetBeans and Eclipse can greatly simplify the process for creating and running test classes. This section introduces using JUnit from NetBeans, and the next section will introduce using JUnit from Eclipse.

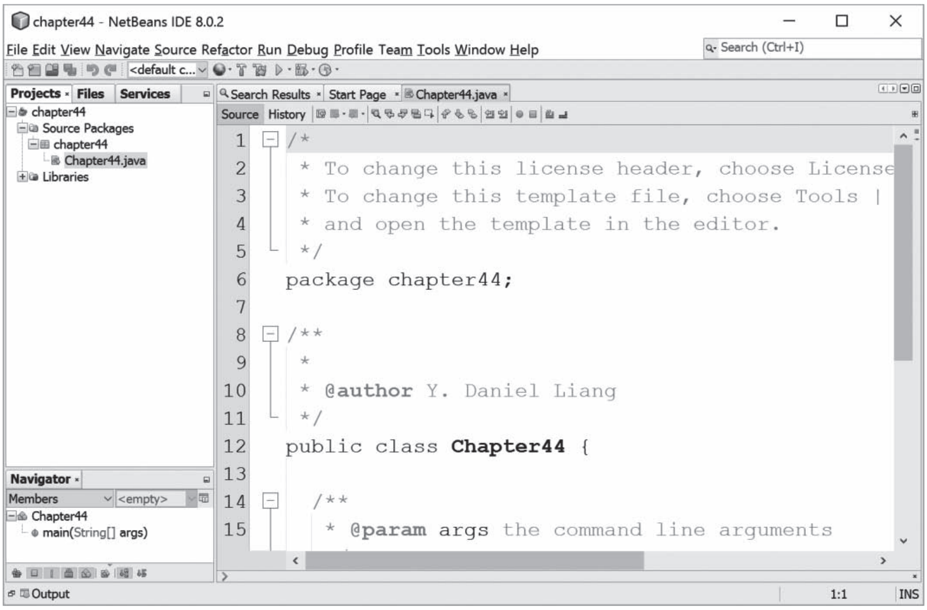
If you are not familiar with NetBeans, see Supplement II.B. Assume you have installed NetBeans 8 or higher. Create a project named `chapter44` as follows:

Step 1: Choose *File*, *New Project* to display the New Project dialog box.

Step 2: Choose Java in the Categories section and Java Application in the Projects section. Click *Next* to display the New Java Application dialog box.

Step 3: Enter `chapter44` as the Project Name and `c:\book` as Project Location. Click *Finish* to create the project as shown in Figure 44.6.

To demonstrate how to create a test class, we first create a class to be tested. Let the class be `Loan` from Listing 10.2. Here are the steps to create the `Loan` class under `chapter44`.



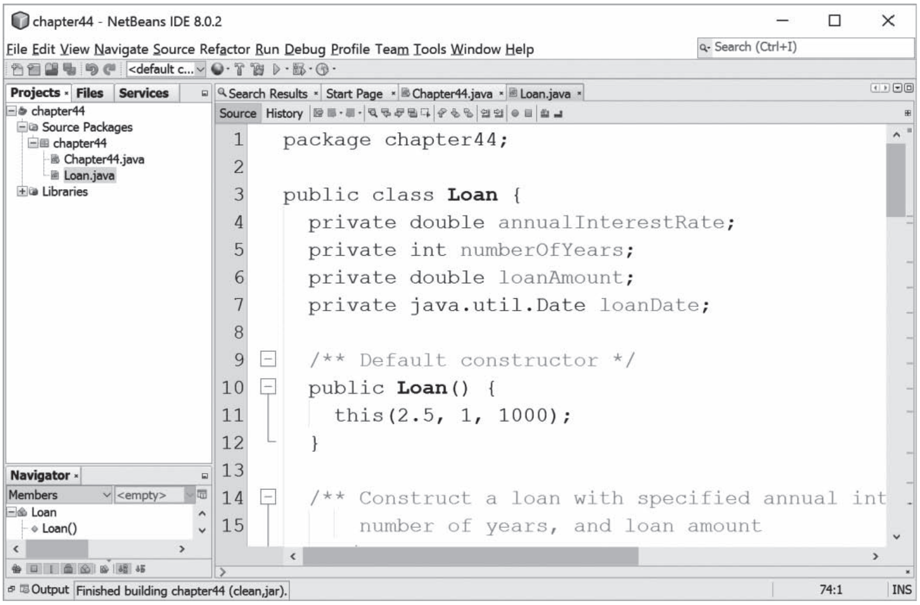
```
/*
 * To change this license header, choose License
 * To change this template file, choose Tools |
 * and open the template in the editor.
 */
package chapter44;

/**
 *
 * @author Y. Daniel Liang
 */
public class Chapter44 {

  /**
   * @param args the command line arguments
```

FIGURE 44.6 A new project named `chapter44` is created.

Step 1: Right-click the project node chapter44 and choose *New*, *Java Class* to display the New Java Class dialog box.

Step 2: Enter Loan as Class Name and chapter44 in the Package field and click *Finish* to create the class.

Step 3: Copy the code in Listing 10.2 to the Loan class and make sure the first line is package chapter44, as shown in Figure 44.7.



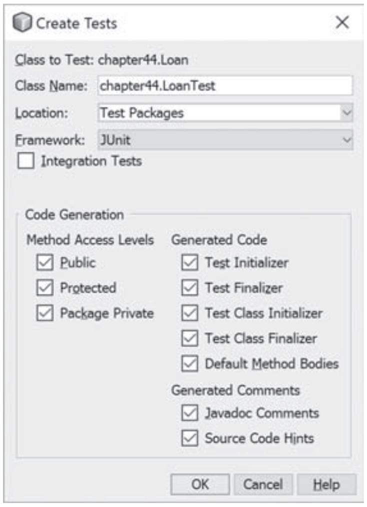
```
package chapter44;

public class Loan {
  private double annualInterestRate;
  private int numberOfYears;
  private double loanAmount;
  private java.util.Date loanDate;

  /** Default constructor */
  public Loan() {
    this(2.5, 1, 1000);
  }

  /** Construct a loan with specified annual int
     number of years, and loan amount
```

Finished building chapter44 (clean,jar).

FIGURE 44.7 The Loan class is created.

Now you can create a test class to test the Loan class as follows:

Step 1: Right-click Loan.java in the project to display a context menu and choose *Tools*, *Create/Update Test* to display the Create Test dialog box, as shown in Figure 44.8.



Class to Test: chapter44.Loan

Class Name: chapter44.LoanTest

Location: Test Packages

Framework: JUnit

□ Integration Tests

Code Generation

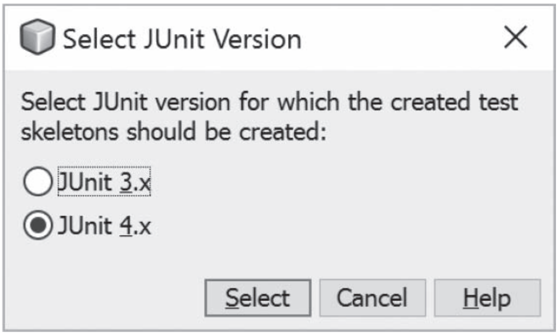
Method Access Levels
- ☑ Public
- ☑ Protected
- ☑ Package Private

Generated Code
- ☑ Test Initializer
- ☑ Test Finalizer
- ☑ Test Class Initializer
- ☑ Test Class Finalizer
- ☑ Default Method Bodies

Generated Comments
- ☑ Javadoc Comments
- ☑ Source Code Hints

FIGURE 44.8 The Create Tests dialog box creates a Test class.

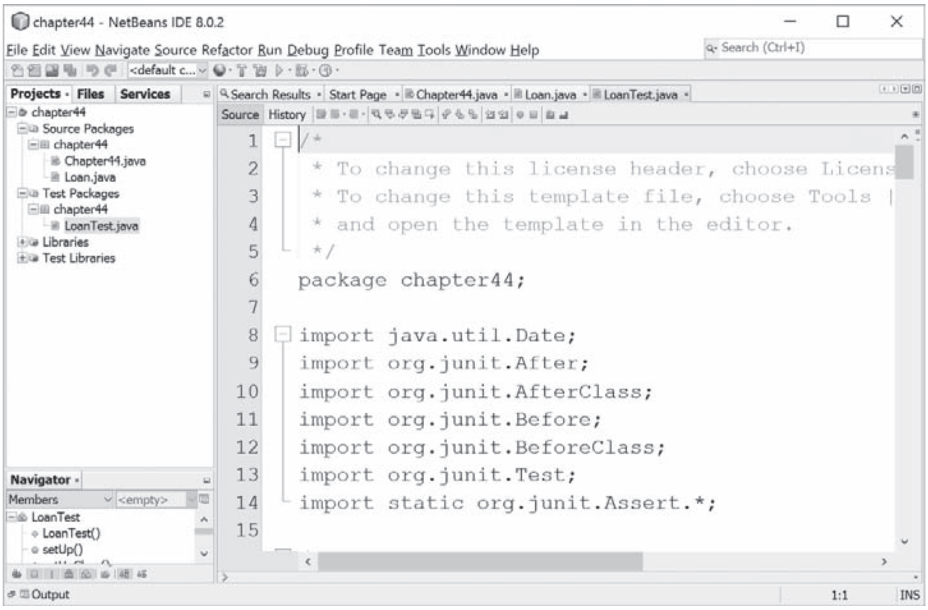
Step 2: Click OK. You will see the Select JUnit version dialog box displayed as shown in Figure 44.9. Choose Junit 4.x. Click *OK* to generate a test class named `LoanTest` as shown in Figure 44.10. Note that LoanTest.java is placed under the Test Packages node in the project.



Select JUnit version for which the created test skeletons should be created:

○ JUnit 3.x
● JUnit 4.x

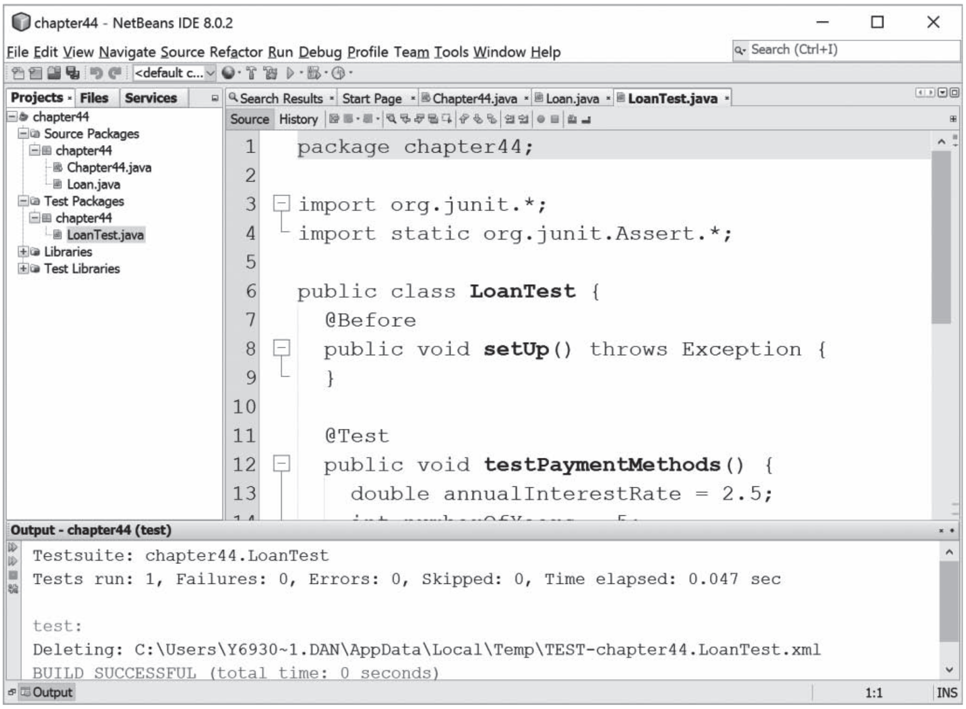
FIGURE 44.9 You should select JUnit 4.x framework to create test classes.



```
/*
 * To change this license header, choose Licens
 * To change this template file, choose Tools |
 * and open the template in the editor.
 */
package chapter44;

import java.util.Date;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

```

FIGURE 44.10 The `LoanTest` class is automatically generated.

You can now modify `LoanTest` by copying the code from Listing 44.2. Run LoanTest.java. You will see the test report as shown in Figure 44.11.



```
package chapter44;

import org.junit.*;
import static org.junit.Assert.*;

public class LoanTest {
  @Before
  public void setUp() throws Exception {
  }

  @Test
  public void testPaymentMethods() {
    double annualInterestRate = 2.5;
    int numberOfYears = 5;
```

```
Testsuite: chapter44.LoanTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.047 sec

test:
Deleting: C:\Users\Y6930~1.DAN\AppData\Local\Temp\TEST-chapter44.LoanTest.xml
BUILD SUCCESSFUL (total time: 0 seconds)
```

FIGURE 44.11 The test report is displayed after the `LoanTest` class is executed.

44.4 Using JUnit from Eclipse

Key Point

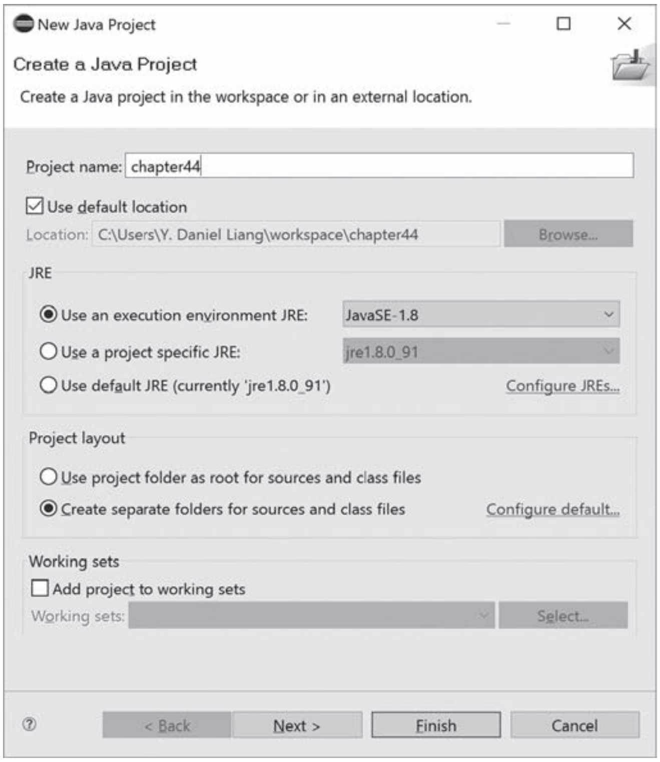
JUnit is intergrated with Eclipse. Using Eclipse, the test program can be automatically generated and the test process can be automated.

This section introduces using JUnit from Eclipse. If you are not familiar with Eclipse, see Supplement II.D. Assume you have installed Eclipse 4.5 or higher. Create a project named `chapter50` as follows:

Step 1: Choose *File*, *New Java Project* to display the New Java Project dialog box, as shown in Figure 44.12.

Step 2: Enter `chapter50` in the project name field and click *Finish* to create the project.

To demonstrate how to create a test class, we first create a class to be tested. Let the class be `Loan` from Listing 10.2. Here are the steps to create the `Loan` class under `chapter44`.



New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: chapter44

☑ Use default location

Location: C:\Users\Y. Daniel Liang\workspace\chapter44

JRE

● Use an execution environment JRE: JavaSE-1.8

○ Use a project specific JRE: jre1.8.0_91

○ Use default JRE (currently 'jre1.8.0_91')

Project layout

○ Use project folder as root for sources and class files

● Create separate folders for sources and class files

Working sets

□ Add project to working sets

Working sets:

FIGURE 44.12 The New Java Project dialog creates a new project.

Step 1: Right-click the project node `chapter44` and choose *New*, *Class* to display the New Java Class dialog box, as shown in Figure 44.13.

Step 2: Enter `mytest` in the Package field and click *Finish* to create the class.

Step 3: Copy the code in Listing 10.2 to the `Loan` class and make sure the first line is `package mytest`, as shown in Figure 44.14.

Now you can create a test class to test the `Loan` class as follows:

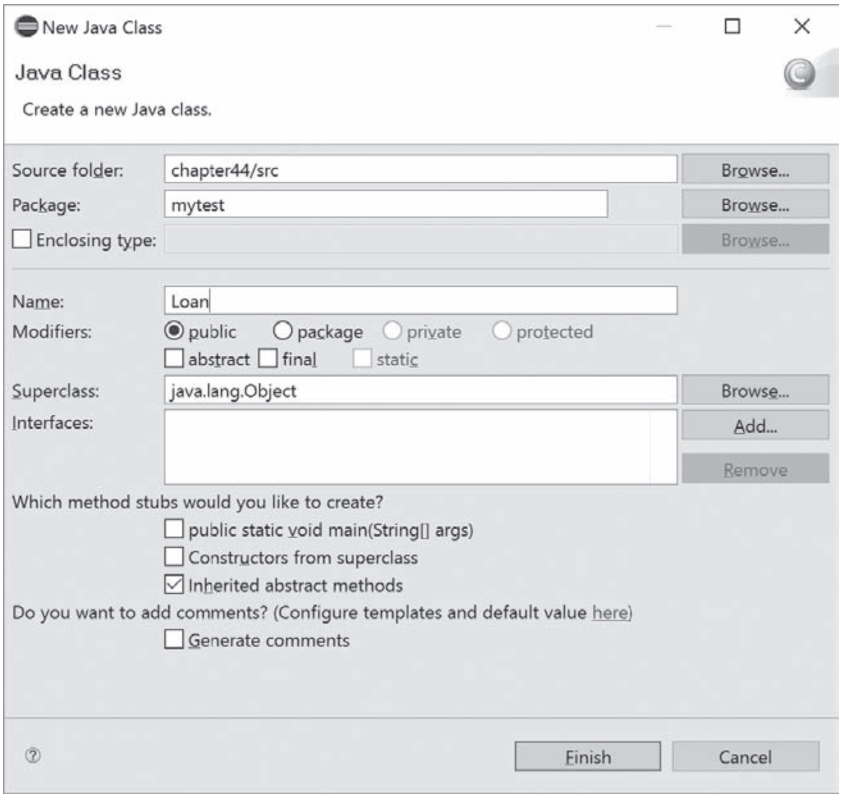
Step 1: Right-click Loan.java in the project to display a context menu and choose *New*, *JUnit Test Case* to display the New JUnit Test Case dialog box, as shown in Figure 44.15.

Step 2: Click *Finish*. You will see a dialog prompting you to add JUnit 4 to the project build path. Click *OK* to add it. Now a test class named LoanTest is created as shown in Figure 44.16.

You can now modify `LoanTest` by copying the code from Listing 44.2. Run LoanTest.java. You will see the test report as shown in Figure 44.17.

KEY TERMS

JUnit	44-2	test class	44-2
JUnitCore	44-2	test runner	44-2



Source folder: chapter44/src

Package: mytest

□ Enclosing type:

Name: Loan

Modifiers: ● public ○ package ○ private ○ protected

□ abstract □ final □ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

□ public static void main(String[] args)

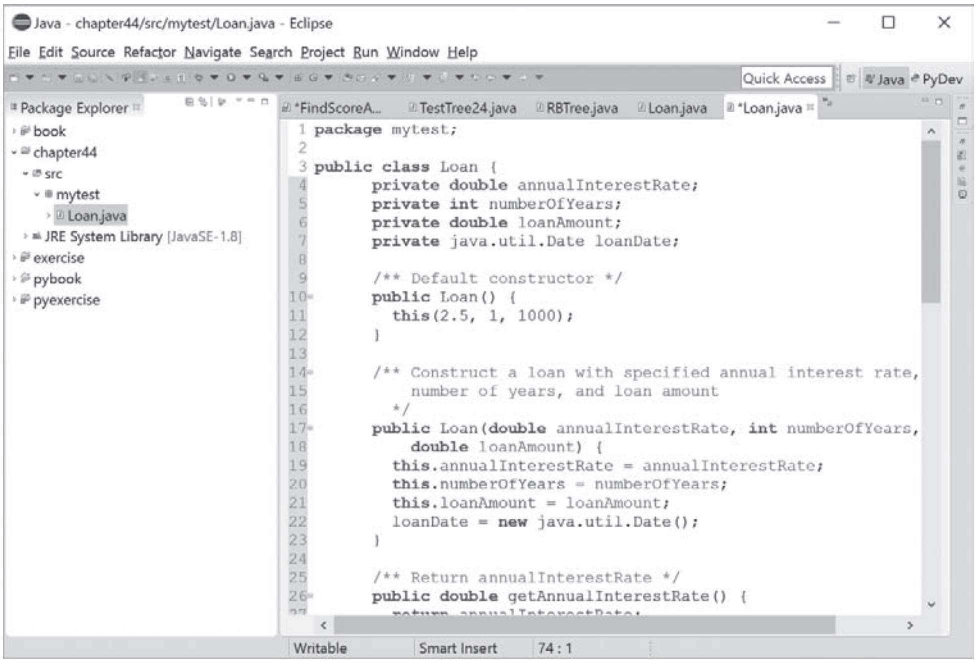
□ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

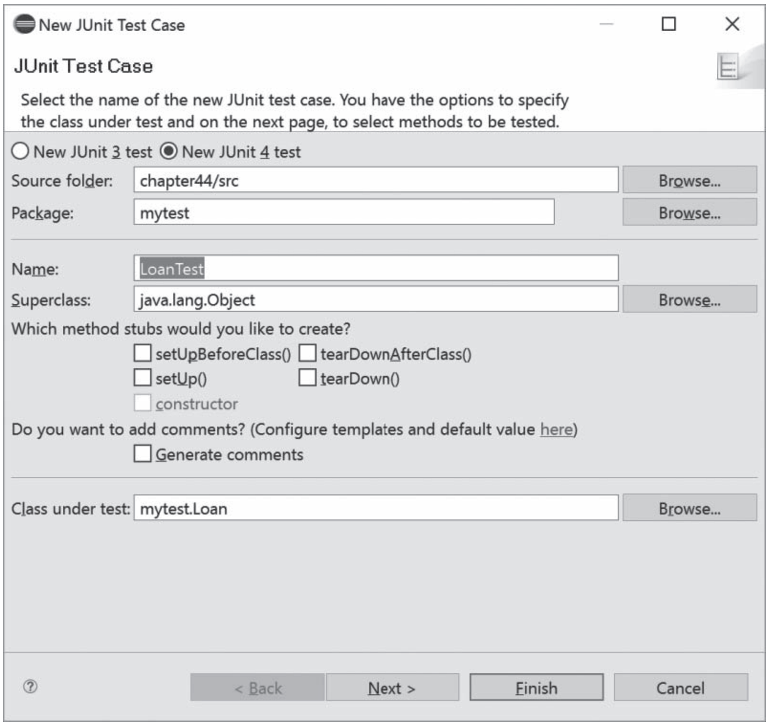
□ Generate comments

FIGURE 44.13 The New Java Class dialog creates a new Java class.



```
package mytest;

public class Loan {
        private double annualInterestRate;
        private int numberOfYears;
        private double loanAmount;
        private java.util.Date loanDate;

        /** Default constructor */
        public Loan() {
          this(2.5, 1, 1000);
        }

        /** Construct a loan with specified annual interest rate,
            number of years, and loan amount
          */
        public Loan(double annualInterestRate, int numberOfYears,
            double loanAmount) {
          this.annualInterestRate = annualInterestRate;
          this.numberOfYears = numberOfYears;
          this.loanAmount = loanAmount;
          loanDate = new java.util.Date();
        }

        /** Return annualInterestRate */
        public double getAnnualInterestRate() {
```

FIGURE 44.14 The Loan class is created.



New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

○ New JUnit 3 test ● New JUnit 4 test

Source folder: chapter44/src

Package: mytest

Name: LoanTest

Superclass: java.lang.Object

Which method stubs would you like to create?

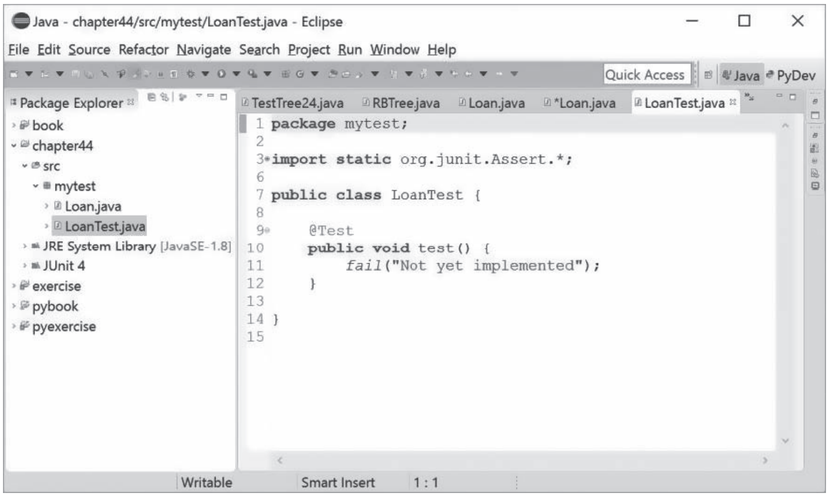
□ setUpBeforeClass() □ tearDownAfterClass()
□ setUp() □ tearDown()
□ constructor

Do you want to add comments? (Configure templates and default value here)

□ Generate comments

Class under test: mytest.Loan

FIGURE 44.15 The New JUnit Test Case dialog box creates a Test class.

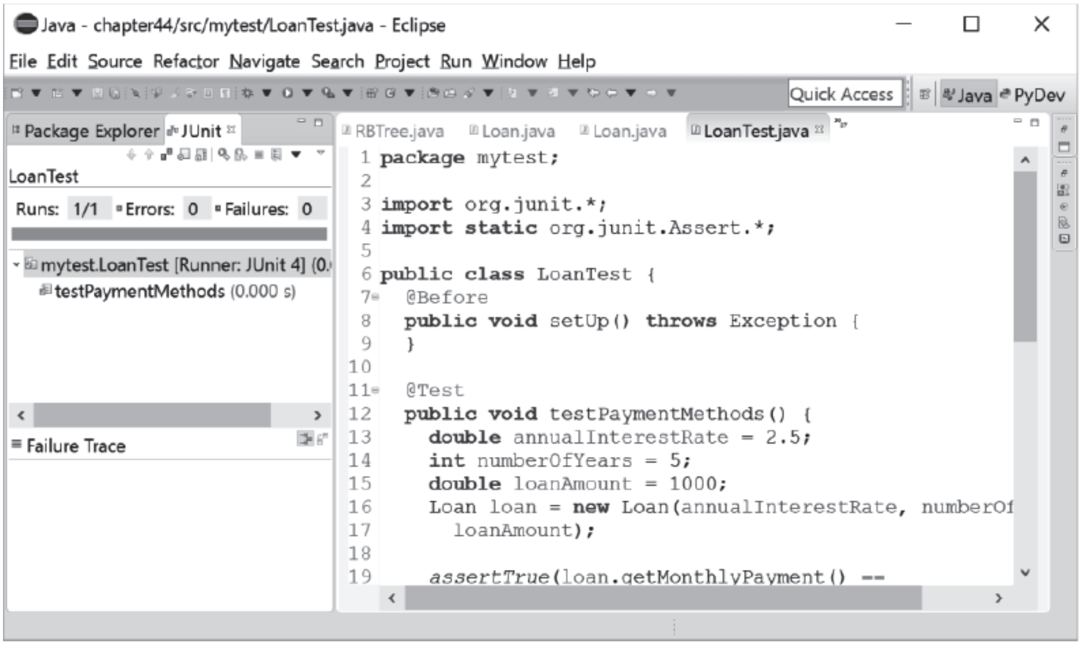


Java - chapter44/src/mytest/LoanTest.java - Eclipse

- book
- chapter44
  - src
    - mytest
      - Loan.java
      - LoanTest.java
  - JRE System Library [JavaSE-1.8]
  - JUnit 4
- exercise
- pybook
- pyexercise

```
package mytest;

import static org.junit.Assert.*;

public class LoanTest {

    @Test
    public void test() {
        fail("Not yet implemented");
    }

}

```

FIGURE 44.16 The LoanTest class is automatically generated.


Java - chapter44/src/mytest/LoanTest.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
PyDev
Package Explorer
JUnit
RBTree.java
Loan.java
Loan.java
LoanTest.java

LoanTest

Runs: 1/1 Errors: 0 Failures: 0

mytest.LoanTest [Runner: JUnit 4] (0.
testPaymentMethods (0.000 s)

Failure Trace

```
package mytest;

import org.junit.*;
import static org.junit.Assert.*;

public class LoanTest {
  @Before
  public void setUp() throws Exception {
  }

  @Test
  public void testPaymentMethods() {
    double annualInterestRate = 2.5;
    int numberOfYears = 5;
    double loanAmount = 1000;
    Loan loan = new Loan(annualInterestRate, numberOf
      loanAmount);

    assertTrue(loan.getMonthlyPayment() ==
```

Figure 44.17 The test report is displayed after the `LoanTest` class is executed.

Chapter Summary

1. JUnit is an open-source framework for testing Java programs.

2. To test a Java class, you create a test class for the class to be tested and use JUnit's test runner to execute the test class to generate a test report.

3. You can create and run a test class from the command window or use a tool such as NetBeans and Eclipse.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

Programming Exercises

44.1 Write a test class to test the methods `length`, `charAt`, `substring`, and `indexOf` in the `java.lang.String` class.

44.2 Write a test class to test the methods `add`, `remove`, `addAll`, `removeAll`, `size`, `isEmpty`, and `contains` in the `java.util.HashSet` class.

44.3 Write a test class to test the method `isPrime` in Listing 6.7, PrimeNumberMethod.java.

44.4 Write a test class to test the methods `getBMI` and `getStatus` in the BMI class in Listing 10.4.