Here, *radius* specifies the radius of the shadow inside the node. In essence, the radius describes the size of the shadow. The color of the shadow is specified by *shadowColor*. Here, the type **Color** is the JavaFX type **javafx.scene.paint.Color**. It defines a large number of constants, such as **Color.GREEN**, **Color.RED**, and **Color.BLUE**, which makes it easy to use.

Transforms

Transforms are supported by the abstract **Transform** class, which is packaged in **javafx.scene.transform**. Four of its subclasses are **Rotate**, **Scale**, **Shear**, and **Translate**. Each does what its name suggests. (Another subclass is **Affine**, but typically you will use one or more of the preceding transform classes.) It is possible to perform more than one transform on a node. For example, you could rotate it and scale it. Transforms are supported by the **Node** class as described next.

One way to add a transform to a node is to add it to the list of transforms maintained by the node. This list is obtained by calling **getTransforms( )**, which is defined by **Node**. It is shown here:

```
final ObservableList<Transform> getTransforms( )
```

It returns a reference to the list of transforms. To add a transform, simply add it to this list by calling **add( )**. You can clear the list by calling **clear( )**. You can use **remove( )** to remove a specific element.

In some cases, you can specify a transform directly, by setting one of **Node**'s properties. For example, you can set the rotation angle of a node, with the pivot point being at the center of the node, by calling **setRotate( )**, passing in the desired angle. You can set a scale by using **setScaleX( )** and **setScaleY( )**, and you can translate a node by using **setTranslateX( )** and **setTranslateY( )**. (*Z* axis translations may also be supported by the platform.) However, using the transforms list offers the greatest flexibility, and that is the approach demonstrated here.

NOTE Any transforms specified on a node directly will be applied after all transforms in the transforms list.

To demonstrate the use of transforms, we will use the **Rotate** and **Scale** classes. The other transforms are used in the same general way. **Rotate** rotates a node around a specified point. It defines several constructors. Here is one example:

```
Rotate(double angle, double x, double y)
```

Here, *angle* specifies the number of degrees to rotate. The center of rotation, called the *pivot point*, is specified by *x* and *y*. It is also possible to use the default constructor and set these values after a **Rotate** object has been created, which is what the following demonstration program will do. This is done by using the **setAngle( )**, **setPivotX( )**, and **setPivotY( )** methods, shown here:

```
final void setAngle(double angle)
final void setPivotX(double x)
final void setPivotY(double y)
```

As before, *angle* specifies the number of degrees to rotate and the center of rotation is specified by *x* and *y*.

Scale scales a node as specified by a scale factor. **Scale** defines several constructors. This is the one we will use:

Scale(double *widthFactor*, double *heightFactor*)

Here, *widthFactor* specifies the scaling factor applied to the node's width, and *heightFactor* specifies the scaling factor applied to the node's height. These factors can be changed after a **Scale** instance has been created by using **setX( )** and **setY( )**, shown here:

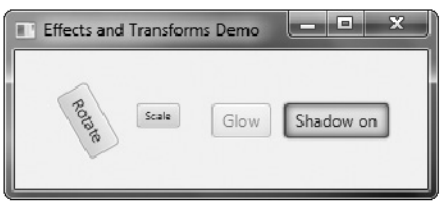
final void setX(double *widthFactor*)

final void setY(double *heightFactor*)

As before, *widthFactor* specifies the scaling factor applied to the node's width, and *heightFactor* specifies the scaling factor applied to the node's height.

Demonstrating Effects and Transforms

The following program demonstrates the use of effects and transforms. It does so by creating four buttons called Rotate, Scale, Glow, and Shadow. Each time one of these buttons is pressed, the corresponding effect or transform is applied to the button. Sample output is shown here:



When you examine the program, you will see how easy it is to customize the look of your GUI. You might find it interesting to experiment with it, trying different transforms or effects, or trying the effects on different types of nodes other than buttons.

```
// Demonstrate rotation, scaling, glowing, and inner shadow.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.transform.*;
import javafx.scene.effect.*;
import javafx.scene.paint.*;

public class EffectsAndTransformsDemo extends Application {
```

```
double angle = 0.0;
double glowVal = 0.0;
boolean shadow = false;
double scaleFactor = 1.0;

// Create initial effects and transforms.
Glow glow = new Glow(0.0);
InnerShadow innerShadow = new InnerShadow(10.0, Color.RED);
Rotate rotate = new Rotate();
Scale scale = new Scale(scaleFactor, scaleFactor);

// Create four push buttons.
Button btnRotate = new Button("Rotate");
Button btnGlow = new Button("Glow");
Button btnShadow = new Button("Shadow off");
Button btnScale = new Button("Scale");

public static void main(String[] args) {

  // Start the JavaFX application by calling launch().
  launch(args);
}

// Override the start() method.
public void start(Stage myStage) {

  // Give the stage a title.
  myStage.setTitle("Effects and Transforms Demo");

  // Use a FlowPane for the root node. In this case,
  // vertical and horizontal gaps of 10 are used.
  FlowPane rootNode = new FlowPane(10, 10);

  // Center the controls in the scene.
  rootNode.setAlignment(Pos.CENTER);

  // Create a scene.
  Scene myScene = new Scene(rootNode, 300, 100);

  // Set the scene on the stage.
  myStage.setScene(myScene);

  // Set the initial glow effect.
  btnGlow.setEffect(glow);

  // Add rotation to the transform list for the Rotate button.
  btnRotate.getTransforms().add(rotate);

  // Add scaling to the transform list for the Scale button.
  btnScale.getTransforms().add(scale);

  // Handle the action events for the Rotate button.
  btnRotate.setOnAction(new EventHandler<ActionEvent>() {
```

```
  public void handle(ActionEvent ae) {
    // Each time button is pressed, it is rotated 30 degrees
    // around its center.
    angle += 30.0;

    rotate.setAngle(angle);
    rotate.setPivotX(btnRotate.getWidth()/2);
    rotate.setPivotY(btnRotate.getHeight()/2);
  }
});

// Handle the action events for the Scale button.
btnScale.setOnAction(new EventHandler<ActionEvent>() {
  public void handle(ActionEvent ae) {
    // Each time button is pressed, the button's scale is changed.
    scaleFactor += 0.1;
    if(scaleFactor > 1.0) scaleFactor = 0.4;

    scale.setX(scaleFactor);
    scale.setY(scaleFactor);

  }
});


// Handle the action events for the Glow button.
btnGlow.setOnAction(new EventHandler<ActionEvent>() {
  public void handle(ActionEvent ae) {
    // Each time button is pressed, its glow value is changed.
    glowVal += 0.1;
    if(glowVal > 1.0) glowVal = 0.0;

    // Set the new glow value.
    glow.setLevel(glowVal);
  }
});

// Handle the action events for the Shadow button.
btnShadow.setOnAction(new EventHandler<ActionEvent>() {
  public void handle(ActionEvent ae) {
    // Each time button is pressed, its shadow status is changed.
    shadow = !shadow;
    if(shadow) {
      btnShadow.setEffect(innerShadow);
      btnShadow.setText("Shadow on");
    } else {
      btnShadow.setEffect(null);
      btnShadow.setText("Shadow off");
    }
  }
});

// Add the label and buttons to the scene graph.
rootNode.getChildren().addAll(btnRotate, btnScale, btnGlow, btnShadow);
```

```
    // Show the stage and its scene.
    myStage.show();
  }
}
```

Before leaving the topic of effects and transforms, it is useful to mention that several of them are particularly pleasing when used on a **Text** node. **Text** is a class packaged in **javafx.scene.text**. It creates a node that consists of text. Because it is a node, the text can be easily manipulated as a unit and various effects and transforms can be applied.

Adding Tooltips

One very popular element in the modern GUI is the *tooltip*. A tooltip is a short message that is displayed when the mouse hovers over a control. In JavaFX, a tooltip can be easily added to any control. Frankly, because of the benefits that tooltips offer and the ease by which they can be incorporated into your GUI, there is virtually no reason not to use them where appropriate.

To add a tooltip, you call the **setTooltip( )** method defined by **Control**. (**Control** is a base class for all controls.) The **setTooltip( )** method is shown here:

final void setTooltip(Tooltip *tip*)

In this case, *tip* is an instance of **Tooltip**, which specifies the tooltip. Once a tooltip has been set, it is automatically displayed when the mouse hovers over the control. No other action is required on your part.

The **Tooltip** class encapsulates a tooltip. This is the constructor that we will use:

Tooltip(String *str*)

Here, *str* specifies the message that will be displayed by the tooltip.

To see tooltips in action, try adding the following statements to the **CheckboxDemo** program shown earlier.

```
cbWeb.setTooltip(new Tooltip("Deploy to Web"));
cbDesktop.setTooltip(new Tooltip("Deploy to Desktop"));
cbMobile.setTooltip(new Tooltip("Deploy to Mobile"));
```

After these additions, the tooltips will be displayed for each check box.

Disabling a Control

Before leaving the subject of controls, it is useful to describe one more feature. Any node in the scene graph, including a control, can be disabled under program control. To disable a control, use **setDisable( )**, defined by **Node**. It is shown here:

final void setDisable(boolean *disable*)

If *disable* is **true**, the control is disabled; otherwise, it is enabled. Thus, using **setDisable( )**, you can disable a control and then enable it later.

CHAPTER

36 Introducing JavaFX Menus

Menus are an important part of many GUIs because they give the user access to a program's core functionally. Furthermore, the proper implementation of an application's menus is a necessary part of creating a successful GUI. Because of the key role they play in many applications, JavaFX provides extensive support for menus. Fortunately, JavaFX's approach to menus is both powerful and streamlined.

As you will see throughout the course of this chapter, JavaFX menus have several parallels with Swing menus, which were described in Chapter 33. As a result, if you already know how to create Swing menus, learning how to create menus in JavaFX is easy. That said, there are also several differences, so it is important not to jump to conclusions about the JavaFX menu system.

The JavaFX menu system supports several key elements, including

- The menu bar, which is the main menu for an application.
- The standard menu, which can contain either items to be selected or other menus (submenus).
- The context menu, which is often activated by right-clicking the mouse. Context menus are also called popup menus.

JavaFX menus also support *accelerator keys,* which enable menu items to be selected without having to activate the menu, and *mnemonics,* which allow a menu item to be selected by the keyboard once the menu options are displayed. In addition to "normal" menus, JavaFX also supports the *toolbar*, which provides rapid access to program functionality, often paralleling menu items.

Menu Basics

The JavaFX menu system is supported by a group of related classes packaged in **javafx.scene.control**. The ones used in this chapter are shown in Table 36-1, and they represent the core of the menu system. Although JavaFX allows a high degree of

Class	Description
CheckMenuItem	A check menu item.
ContextMenu	A popup menu that is typically activated by right-clicking the mouse.
Menu	A standard menu. A menu consists of one or more **MenuItem**s.
MenuBar	An object that holds the top-level menu for the application.
MenuItem	An object that populates menus.
RadioMenuItem	A radio menu item.
SeparatorMenuItem	A visual separator between menu items.

Table 36-1 The Core JavaFX Menu Classes

customization if desired, normally you will simply use the menu classes as-is because their default look and feel is generally what you will want.

Here is brief overview of how the classes fit together. To create a main menu for an application, you first need an instance of **MenuBar**. This class is, loosely speaking, a container for menus. To the **MenuBar** you add instances of **Menu**. Each **Menu** object defines a menu. That is, each **Menu** object contains one or more selectable items. The items displayed by a **Menu** are objects of type **MenuItem**. Thus, a **MenuItem** defines a selection that can be chosen by the user.

In addition to "standard" menu items, you can also include check and radio menu items in a menu. Their operation parallels check box and radio button controls. A check menu item is created by **CheckMenuItem**. A radio menu item is created by **RadioMenuItem**. Both of these classes extend **MenuItem**.

SeparatorMenuItem is a convenience class that creates a separator line in a menu. It inherits **CustomMenuItem**, which is a class that facilitates embedding other types of controls in a menu item. **CustomMenuItem** extends **MenuItem**.

One key point about JavaFX menus is that **MenuItem** does *not* inherit **Node**. Thus, instances of **MenuItem** can only be used in a menu. They cannot be otherwise incorporated into a scene graph. However, **MenuBar** does inherit **Node**, which does allow the menu bar to be added to the scene graph.

Another key point is that **MenuItem** is a superclass of **Menu**. This allows the creation of submenus, which are, essentially, menus within menus. To create a submenu, you first create and populate a **Menu** object with **MenuItem**s and then add it to another **Menu** object. You will see this process in action in the examples that follow.

When a menu item is selected, an action event is generated. The text associated with the selection will be the name of the selection. Thus, when using one action event handler to process all menu selections, one way you can determine which item was selected is by examining the name. Of course, you can also use separate anonymous inner classes or lambda expressions to handle each menu item's action events. In this case, the menu selection is already known and there is no need to examine the name to determine which item was selected.

As an alternative or adjunct to menus that descend from the menu bar, you can also create stand-alone, context menus, which pop up when activated. To create a context menu, first create an object of type **ContextMenu**. Then, add **MenuItem**s to it. A context menu is often activated by clicking the right mouse button when the mouse is over a control

for which a context menu has been defined. It is important to point out that **ContextMenu** is not derived from **MenuItem**. Rather, it inherits **PopupControl**.

A feature related to the menu is the *toolbar*. In JavaFX, toolbars are supported by the **ToolBar** class. It creates a stand-alone component that is often used to provide fast access to functionality contained within the menus of the application. For example, a toolbar might provide fast access to the formatting commands supported by a word processor.

An Overview of MenuBar, Menu, and MenuItem

Before you can create a menu, you need to know some specifics about **MenuBar**, **Menu**, and **MenuItem**. These form the minimum set of classes needed to construct a main menu for an application. **MenuItem**s are also used by context (i.e., popup) menus. Thus, these classes form the foundation of the menu system.

MenuBar

MenuBar is essentially a container for menus. It is the control that supplies the main menu of an application. Like all JavaFX controls, it inherits **Node**. Thus, it can be added to a scene graph. **MenuBar** has only one constructor, which is the default constructor. Therefore, initially, the menu bar will be empty, and you will need to populate it with menus prior to use. As a general rule, an application has one and only one menu bar.

MenuBar defines several methods, but often you will use only one: **getMenus( )**. It returns a list of the menus managed by the menu bar. It is to this list that you will add the menus that you create. The **getMenus( )** method is shown here:

```
final ObservableList<Menu> getMenus( )
```

A **Menu** instance is added to this list of menus by calling **add( )**. You can also use **addAll( )** to add two or more **Menu** instances in a single call. The added menus are positioned in the bar from left to right, in the order in which they are added. If you want to add a menu at a specific location, then use this version of **add( )**:

```
void add(int idx, Menu menu)
```

Here, *menu* is added at the index specified by *idx*. Indexing begins at 0, with 0 being the left-most menu.

In some cases, you might want to remove a menu that is no longer needed. You can do this by calling **remove( )** on the **ObservableList** returned by **getMenus( )**. Here are two of its forms:

```
void remove(Menu menu)

void remove(int idx)
```

Here, *menu* is a reference to the menu to remove, and *idx* is the index of the menu to remove. Indexing begins at zero.

It is sometimes useful to obtain a count of the number of items in a menu bar. To do this, call **size( )** on the list returned by **getMenus( )**.

NOTE Recall that **ObservableList** implements the **List** collections interface, which gives you access to all the methods defined by **List**.

Once a menu bar has been created and populated, it is added to the scene graph in the normal way.

Menu

Menu encapsulates a menu, which is populated with **MenuItem**s. As mentioned, **Menu** is derived from **MenuItem**. This means that one **Menu** can be a selection in another **Menu**. This enables one menu to be submenu of another. **Menu** defines three constructors. Perhaps the most commonly used is shown here:

Menu(String *name*)

It creates a menu that has the name specified by *name*. You can specify an image along with text with this constructor:

Menu(String *name*, Node *image*)

Here, *image* specifies the image that is displayed. In all cases, the menu is empty until menu items are added to it. Finally, you don't have to give a menu a name when it is constructed. To create an unnamed menu, you can use the default constructor:

Menu( )

In this case, you can add a name and/or image after the fact by calling **setText( )** or **setGraphic( )**.

Each menu maintains a list of menu items that it contains. To add an item to the menu, add items to this list. To do so, first call **getItems( )**, shown here:

final ObservableList<MenuItem> getItems( )

It returns the list of items currently associated with the menu. To this list, add menu items by calling either **add( )** or **addAll( )**. Among other actions, you can remove an item by calling **remove( )** and obtain the size of the list by calling **size( )**.

One other point: You can add a menu separator to the list of menu items, which is an object of type **SeparatorMenuItem**. Separators help organize long menus by allowing you to group related items together. A separator can also help set off an important item, such as the Exit selection in a menu.

MenuItem

MenuItem encapsulates an element in a menu. This element can be either a selection linked to some program action, such as Save or Close, or it can cause a submenu to be displayed. **MenuItem** defines the following three constructors.

MenuItem( )

MenuItem(String *name*)

MenuItem(String *name*, Node *image*)

The first creates an empty menu item. The second lets you specify the name of the item, and the third enables you to include an image.

A **MenuItem** generates an action event when selected. You can register an action event handler for such an event by calling **setOnAction( )**, just as you did when handling button events. It is shown again for your convenience:

final void setOnAction(EventHandler<ActionEvent> *handler*)

Here, *handler* specifies the event handler. You can fire an action event on a menu item by calling **fire( )**.

MenuItem defines several methods. One that is often useful is **setDisable( )**, which you can use to enable or disable a menu item. It is shown here:

final void setDisable(boolean *disable*)

If *disable* is **true**, the menu item is disabled and cannot be selected. If *disable* is **false**, the item is enabled. Using **setDisable( )**, you can turn menu items on or off, depending on program conditions.

Create a Main Menu

As a general rule, the most commonly used menu is the *main menu*. This is the menu defined by the menu bar, and it is the menu that defines all (or nearly all) of the functionality of an application. As you will see, JavaFX streamlines the process of creating and managing the main menu. Here, you will see how to construct a simple main menu. Subsequent sections will show various options.

NOTE As a way of clearly illustrating the similarities and differences between the Swing and JavaFX menu systems, the examples in this chapter rework the menu examples from Chapter 33. If you already know Swing, you might find it helpful to compare the two different approaches.

Constructing the main menu requires several steps. First, create the **MenuBar** instance that will hold the menus. Next, construct each menu that will be in the menu bar. In general, a menu is constructed by first creating a **Menu** object and then adding **MenuItem**s to it. After the menus have been created, add them to the menu bar. Then, the menu bar, itself, must be added to the scene graph. Finally, for each menu item, you must add an action event handler that responds to the action event fired when a menu item is selected.

A good way to understand the process of creating and managing menus is to work through an example. Here is a program that creates a simple menu bar that contains three menus. The first is a standard File menu that contains Open, Close, Save, and Exit selections. The second menu is called Options, and it contains two submenus called Colors and Priority. The third menu is called Help, and it has one item: About. When a menu item is selected, the name of the selection is displayed in a label.

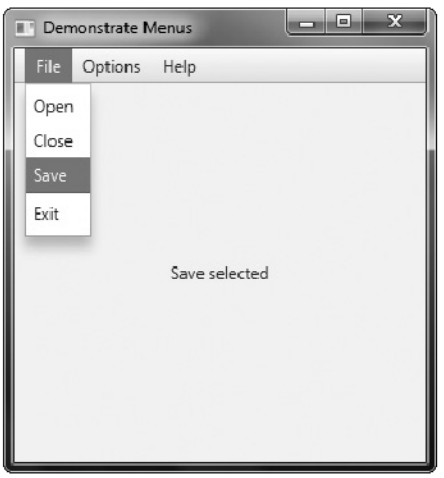
```
// Demonstrate Menus

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
```

```
import javafx.event.*;
import javafx.geometry.*;

public class MenuDemo extends Application {

  Label response;

  public static void main(String[] args) {

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the start() method.
  public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("Demonstrate Menus");

    // Use a BorderPane for the root node.
    BorderPane rootNode = new BorderPane();

     // Create a scene.
    Scene myScene = new Scene(rootNode, 300, 300);

    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Create a label that will report the selection.
    response = new Label("Menu Demo");

    // Create the menu bar.
    MenuBar mb = new MenuBar();

    // Create the File menu.
    Menu fileMenu = new Menu("File");
    MenuItem open = new MenuItem("Open");
    MenuItem close = new MenuItem("Close");
    MenuItem save = new MenuItem("Save");
    MenuItem exit = new MenuItem("Exit");
    fileMenu.getItems().addAll(open, close, save,
                               new SeparatorMenuItem(), exit);

    // Add File menu to the menu bar.
    mb.getMenus().add(fileMenu);

    // Create the Options menu.
    Menu optionsMenu = new Menu("Options");

    // Create the Colors submenu.
    Menu colorsMenu = new Menu("Colors");
    MenuItem red = new MenuItem("Red");
    MenuItem green = new MenuItem("Green");
    MenuItem blue = new MenuItem("Blue");
```

```
colorsMenu.getItems().addAll(red, green, blue);
optionsMenu.getItems().add(colorsMenu);

// Create the Priority submenu.
Menu priorityMenu = new Menu("Priority");
MenuItem high = new MenuItem("High");
MenuItem low = new MenuItem("Low");
priorityMenu.getItems().addAll(high, low);
optionsMenu.getItems().add(priorityMenu);

// Add a separator.
optionsMenu.getItems().add(new SeparatorMenuItem());

// Create the Reset menu item.
MenuItem reset = new MenuItem("Reset");
optionsMenu.getItems().add(reset);

// Add Options menu to the menu bar.
mb.getMenus().add(optionsMenu);

// Create the Help menu.
Menu helpMenu = new Menu("Help");
MenuItem about = new MenuItem("About");
helpMenu.getItems().add(about);

// Add Help menu to the menu bar.
mb.getMenus().add(helpMenu);

// Create one event handler that will handle menu action events.
EventHandler<ActionEvent> MEHandler =
                            new EventHandler<ActionEvent>() {
  public void handle(ActionEvent ae) {
    String name = ((MenuItem)ae.getTarget()).getText();

    // If Exit is chosen, the program is terminated.
    if(name.equals("Exit")) Platform.exit();

    response.setText( name + " selected");
  }
};

// Set action event handlers for the menu items.
open.setOnAction(MEHandler);
close.setOnAction(MEHandler);
save.setOnAction(MEHandler);
exit.setOnAction(MEHandler);
red.setOnAction(MEHandler);
green.setOnAction(MEHandler);
blue.setOnAction(MEHandler);
high.setOnAction(MEHandler);
low.setOnAction(MEHandler);
reset.setOnAction(MEHandler);
about.setOnAction(MEHandler);
```

```
    // Add the menu bar to the top of the border pane and
    // the response label to the center position.
    rootNode.setTop(mb);
    rootNode.setCenter(response);

    // Show the stage and its scene.
    myStage.show();
  }
}
```

Sample output is shown here:



Let's examine, in detail, how the menus in this program are created. First, note that **MenuDemo** uses a **BorderPane** instance for the root node. **BorderPane** is similar to the AWT's **BorderLayout** discussed in Chapter 26. It defines a window that has five areas: top, bottom, left, right, and center. The following methods set the node assigned to these areas:

final void setTop(Node *node*)

final void setBottom(Node *node*)

final void setLeft(Node *node*)

final void setRight(Node *node*)

final void setCenter(Node *node*)

Here, *node* specifies the element, such as a control, that will be shown in each location. Later in the program, the menu bar is positioned in the top location and a label that displays the menu selection is set to the center position. Setting the menu bar to the top position ensures that it will be shown at the top of the application and will automatically be resized to fit the horizontal width of the window. This is why **BorderPane** is used in the menu examples. Of course, other approaches, such as using a **VBox**, are also valid.

Much of the code in the program is used to construct the menu bar, its menus, and menu items, and this code warrants a close inspection. First, the menu bar is constructed and a reference to it is assigned to **mb** by this statement:

```
// Create the menu bar.
MenuBar mb = new MenuBar();
```

At this point, the menu bar is empty. It will be populated by the menus that follow.

Next, the File menu and its menu entries are created by this sequence:

```
// Create the File menu.
Menu fileMenu = new Menu("File");
MenuItem open = new MenuItem("Open");
MenuItem close = new MenuItem("Close");
MenuItem save = new MenuItem("Save");
MenuItem exit = new MenuItem("Exit");
```

The names Open, Close, Save, and Exit will be shown as selections in the menu. The menu entries are added to the File menu by this call to **addAll( )** on the list of menu items returned by **getItems( )**:

```
fileMenu.getItems().addAll(open, close, save,
                           new SeparatorMenuItem(), exit);
```

Recall that **getItems( )** returns the menu items associated with a **Menu** instance. To add menu items to a menu, you will add them to this list. Notice that a separator is used to separate visually the Exit entry from the others.

Finally, the File menu is added to the menu bar by this line:

```
// Add File menu to the menu bar.
mb.getMenus().add(fileMenu);
```

Once the preceding code sequence completes, the menu bar will contain one entry: File. The File menu will contain four selections in this order: Open, Close, Save, and Exit.

The Options menu is constructed using the same basic process as the File menu. However, the Options menu consists of two submenus, Colors and Priority, and a Reset entry. The submenus are first constructed individually and then added to the Options menu. As explained, because **Menu** inherits **MenuItem**, a **Menu** can be added as an entry into another **Menu**. This is the way the submenus are created. The Reset item is added last. Then, the Options menu is added to the menu bar. The Help menu is constructed using the same process.

After all of the menus have been constructed, an **ActionEvent** handler called **MEHandler** is created that will process menu selections. For demonstration purposes, a single handler will process all selections, but in a real-world application, it is often easier to specify a separate handler for each individual selection by using anonymous inner classes or lambda expressions. The **ActionEvent** handler for the menu items is shown here:

```
// Create one event handler that will handle all menu events.
EventHandler<ActionEvent> MEHandler = new EventHandler<ActionEvent>() {
```

```
  public void handle(ActionEvent ae) {
    String name = ((MenuItem)ae.getTarget()).getText();

    // If Exit is chosen, the program is terminated.
    if(name.equals("Exit")) Platform.exit();

    response.setText( name + " selected");
  }
};
```

Inside **handle( )**, the target of the event is obtained by calling **getTarget( )**. The returned reference is cast to **MenuItem**, and its name is returned by calling **getText( )**. This string is then assigned to **name**. If **name** contains the string "Exit", the application is terminated by calling **Platform.exit( )**. Otherwise, the name is displayed in the **response** label.

Before continuing, it must be pointed out that a JavaFX application must call **Platform.exit( )**, not **System.exit( )**. The **Platform** class is defined by JavaFX and packaged in **javafx.application**. Its **exit( )** method causes the **stop( )** life-cycle method to be called. **System.exit( )** does not.

Finally, **MEHandler** is registered as the action event handler for each menu item by the following statements:

```
// Set action event handlers for the menu items.
open.setOnAction(MEHandler);
close.setOnAction(MEHandler);
save.setOnAction(MEHandler);
exit.setOnAction(MEHandler);
red.setOnAction(MEHandler);
green.setOnAction(MEHandler);
blue.setOnAction(MEHandler);
high.setOnAction(MEHandler);
low.setOnAction(MEHandler);
reset.setOnAction(MEHandler);
about.setOnAction(MEHandler);
```

Notice that no listeners are added to the Colors or Priority items because they are not actually selections. They simply activate submenus.

Finally, the menu bar is added to the root node by the following line:

```
rootNode.setTop(mb);
```

This causes the menu bar to be placed at the top of the window.

At this point, you might want to experiment a bit with the **MenuDemo** program. Try adding another menu or adding additional items to an existing menu. It is important that you understand the basic menu concepts before moving on because this program will evolve throughout the remainder of this chapter.

Add Mnemonics and Accelerators to Menu Items

The menu created in the preceding example is functional, but it is possible to make it better. In real applications, a menu usually includes support for keyboard shortcuts. These come in two forms: accelerators and mnemonics. An accelerator is a key combination that

lets you select a menu item without having to first activate the menu. As it applies to menus, a mnemonic defines a key that lets you select an item from an active menu by typing the key. Thus, a mnemonic allows you to use the keyboard to select an item from a menu that is already being displayed.

An accelerator can be associated with a **Menu** or **MenuItem**. It is specified by calling **setAccelerator( )**, shown next:

final void setAccelerator(KeyCombination *keyComb*)

Here, *keyComb* is the key combination that is pressed to select the menu item. **KeyCombination** is class that encapsulates a key combination, such as CTRL-S. It is packaged in **javafx.scene.input**.

KeyCombination defines two **protected** constructors, but often you will use the **keyCombination( )** factory method, shown here:

static KeyCombination keyCombination(String *keys*)

In this case, *keys* is a string that specifies the key combination. It typically consists of a modifier, such as CTRL, ALT, SHIFT, or META, and a letter, such as S. There is a special value, called **shortcut**, which can be used to specify the CTRL key in a Windows system and the META key on a Mac. (It also maps to the typically used shortcut key on other types of systems.) Therefore, if you want to specify CTRL-S as the key combination for Save, then use the string "shortcut+S". This way, it will work for both Windows and Mac and elsewhere.

The following sequence adds accelerators to the File menu created by the **MenuDemo** program in the previous section. After making this change, you can directly select a File menu option by pressing CTRL-O, CTRL-C, CTRL-S, or CTRL-E.

```
// Add keyboard accelerators for the File menu.
open.setAccelerator(KeyCombination.keyCombination("shortcut+O"));
close.setAccelerator(KeyCombination.keyCombination("shortcut+C"));
save.setAccelerator(KeyCombination.keyCombination("shortcut+S"));
exit.setAccelerator(KeyCombination.keyCombination("shortcut+E"));
```

A mnemonic can be specified for both **MenuItem** and **Menu** objects, and it is very easy to do. Simply precede the letter in the name of the menu or menu item with an underscore. For example, in the preceding example, to add the mnemonic *F* to the File menu, declare **fileMenu** as shown here:

```
Menu fileMenu = new Menu("_File"); // now defines a mnemonic
```

After making this change, you can select the File menu by typing ALT then F. However, mnemonics are active only if mnemonic parsing is **true** (as it is by default). You can turn mnemonic parsing on or off by using **setMnemonicParsing( ),** shown here:

final void setMnemonicParsing(boolean *enable*)

In this case, if *enable* is **true**, then mnemonic parsing is turned on. Otherwise, it is turned off.

After making these changes, the File menu will now look like this:

Add Images to Menu Items

You can add images to menu items or use images instead of text. The easiest way to add an image is to specify it when the menu item is being constructed using this constructor:

MenuItem(String *name*, Node *image*)

It creates a menu item with the name specified by *name* and the image specified by *image*. For example, here the About menu item is associated with an image when it is created.

```
ImageView aboutIV = new ImageView("aboutIcon.gif");
MenuItem about = new MenuItem("About", aboutIV);
```

After this addition, the image specified by **aboutIV** will be displayed next to the text "About" when the Help menu is displayed, as shown here:

One last point: You can also add an image to a menu item after the item has been created by calling **setGraphic( )**. This lets you change the image during program execution.

Use RadioMenuItem and CheckMenuItem

Although the type of menu items used by the preceding examples are, as a general rule, the most commonly used, JavaFX defines two others: check menu items and radio menu items. These elements can streamline a GUI by allowing a menu to provide functionality that would otherwise require additional, stand-alone components. Also, sometimes including check or radio menu items simply seems most natural for a specific set of features. Whatever your reason, it is easy to use check and/or radio menu items in menus, and both are examined here.

To add a check menu item to a menu, use **CheckMenuItem**. It defines three constructors, which parallel the ones defined by **MenuItem**. The one used in this chapter is shown here:

CheckMenuItem(String *name*)

Here, *name* specifies the name of the item. The initial state of the item is unchecked. If you want to check a check menu item under program control, call **setSelected( )**, shown here:

final void setSelected(boolean *selected*)

If *selected* is **true**, the menu item is checked. Otherwise, it is unchecked.

Like stand-alone check boxes, check menu items generate action events when their state is changed. Check menu items are especially appropriate in menus when you have options that can be selected and you want to display their selected/deselected status.

A radio menu item can be added to a menu by creating an object of type **RadioMenuItem**. **RadioMenuItem** defines a number of constructors. The one used in this chapter is shown here:

RadioMenuItem(String *name*)

It creates a radio menu item that has the name passed in *name*. The item is not selected. As with the case of check menu items, to select a radio menu item, call **setSelected( )**, passing **true** as an argument.

RadioMenuItem works like a stand-alone radio button, generating both change and action events. Like stand-alone radio buttons, menu radio items must be put into a toggle group in order for them to exhibit mutually exclusive selection behavior.

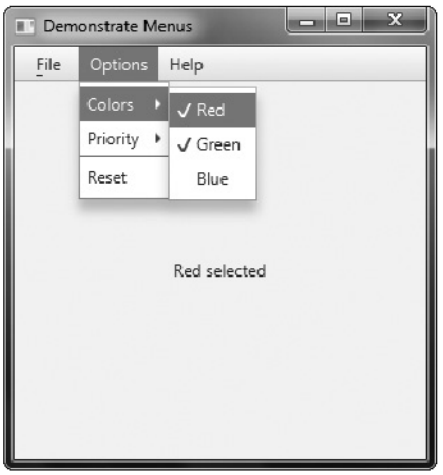
Because both **CheckMenuItem** and **RadioMenuItem** inherit **MenuItem**, each has all of the functionality provided by **MenuItem**. Aside from having the extra capabilities of check boxes and radio buttons, they act like and are used like other menu items.

To try check and radio menu items, first remove the code that creates the Options menu in the **MenuDemo** example program. Then substitute the following code sequence, which uses check menu items for the Colors submenu and radio menu items for the Priority submenu.

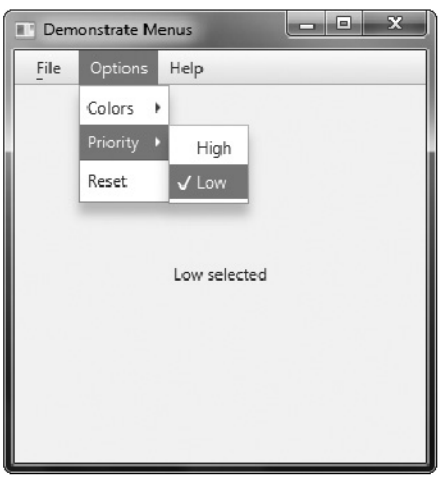
```
// Create the Options menu.
Menu optionsMenu = new Menu("Options");
```

```
// Create the Colors submenu.
Menu colorsMenu = new Menu("Colors");

// Use check menu items for colors. This allows
// the user to select more than one color.
CheckMenuItem red = new CheckMenuItem("Red");
CheckMenuItem green = new CheckMenuItem("Green");
CheckMenuItem blue = new CheckMenuItem("Blue");
colorsMenu.getItems().addAll(red, green, blue);
optionsMenu.getItems().add(colorsMenu);

// Select green for the default color selection.
green.setSelected(true);

// Create the Priority submenu.
Menu priorityMenu = new Menu("Priority");

// Use radio menu items for the priority setting.
// This lets the menu show which priority is used
// and also ensures that one and only one priority
// can be selected at any one time.
RadioMenuItem high = new RadioMenuItem("High");
RadioMenuItem low = new RadioMenuItem("Low");

// Create a toggle group and use it for the radio menu items.
ToggleGroup tg = new ToggleGroup();
high.setToggleGroup(tg);
low.setToggleGroup(tg);

// Select High priority for the default selection.
high.setSelected(true);

// Add the radio menu items to the Priority menu and
// add the Priority menu to the Options menu.
priorityMenu.getItems().addAll(high, low);
optionsMenu.getItems().add(priorityMenu);

// Add a separator.
optionsMenu.getItems().add(new SeparatorMenuItem());

// Create the Reset menu item.
MenuItem reset = new MenuItem("Reset");
optionsMenu.getItems().add(reset);

// Add Options menu to the menu bar.
mb.getMenus().add(optionsMenu);
```

After making the substitution, the check menu items in the Colors submenu look like those shown here:



Here is how the radio menu items in the Priority submenu now look:



Create a Context Menu

A popular alternative or addition to the menu bar is the popup menu, which in JavaFX is referred to as a *context menu*. Typically, a context menu is activated by clicking the right mouse button when over a control. Popup menus are supported in JavaFX by the **ContextMenu** class. The direct superclass of **ContextMenu** is **PopupControl**. An indirect superclass of **ContextMenu** is **javafx.stage.PopupWindow**, which supplies much of its basic functionality.

ContextMenu has two constructors. The one used in this chapter is shown here:

ContextMenu(MenuItem ... *menuItems*)

Here, *menuItems* specify the menu items that will constitute the context menu. The second **ContextMenu** constructor creates an empty menu to which items must be added.

In general, context menus are constructed like regular menus. Menu items are created and added to the menu. Menu item selections are also handled in the same way: by handling action events. The main difference between a context menu and a regular menu is the activation process.

To associate a context menu with a control is amazingly easy. Simply call **setContextMenu( )** on the control, passing in a reference to the menu that you want to pop up. When you right-click on that control, the associated context menu will be shown. The **setContextMenu( )** method is shown here:

final void setContextMenu(ContextMenu *menu*)

In this case, *menu* specifies the context menu associated with the invoking control.

To demonstrate a context menu, we will add one to the **MenuDemo** program. The context menu will present a standard "Edit" menu that includes the Cut, Copy, and Paste entries. It will be set on a text field control. When the mouse is right-clicked while in the text field, the context menu will pop up. To begin, create the context menu, as shown here:

```
// Create the context menu items
MenuItem cut = new MenuItem("Cut");
MenuItem copy = new MenuItem("Copy");
MenuItem paste = new MenuItem("Paste");

// Create a context (i.e., popup) menu that shows edit options.
final ContextMenu editMenu = new ContextMenu(cut, copy, paste);
```

This sequence begins by constructing the **MenuItem**s that will form the menu. It then creates an instance of **ContextMenu** called **editMenu** that contains the items.

Next, add the action event handler to these menu items, as shown here:

```
cut.setOnAction(MEHandler);
copy.setOnAction(MEHandler);
paste.setOnAction(MEHandler);
```

This finishes the construction of the context menu, but the menu has not yet been associated with a control.

Now, add the following sequence that creates the text field:

```
// Create a text field and set its column width to 20.
TextField tf = new TextField();
tf.setPrefColumnCount(20);
```

Next, set the context menu on the text field:

```
// Add the context menu to the textfield.
tf.setContextMenu(editMenu);
```

Now, when the mouse is right-clicked over the text field, the context menu will pop up.

To add the text field to the program, you must create a flow pane that will hold both the text field and the response label. This pane will then be added to the center of the **BorderPane**. This step is necessary because only one node can be added to any single location within a **BorderPane**. First, remove this line of code:

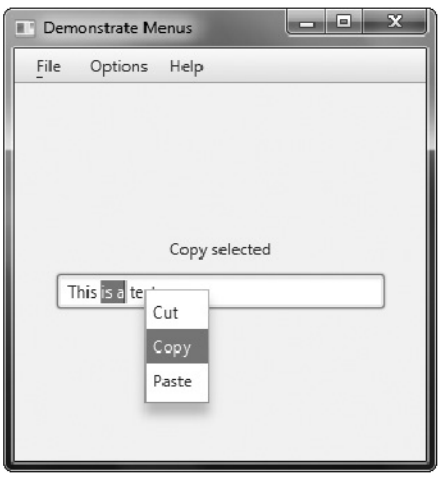
```
rootNode.setCenter(response);
```

Replace it with the following code:

```
// Create a flow pane that will hold both the response
// label and the text field.
FlowPane fpRoot = new FlowPane(10, 10);

// Center the controls in the scene.
fpRoot.setAlignment(Pos.CENTER);

// Add both the label and the text field to the flow pane.
fpRoot.getChildren().addAll(response, tf);

// Add the flow pane to the center of the border layout.
rootNode.setCenter(fpRoot);
```

Of course, the menu bar is still added to the top position of the border pane.

After making these changes, when you right-click over the text field, the context menu will pop up, as shown here:



It is also possible to associate a context menu with a scene. One way to do this is by calling **setOnContextMenuRequested( )** on the root node of the scene. This method is defined by **Node** and is shown here:

final void setOnContextMenuRequested(
EventHandler<? super ContextMenuEvent> *eventHandler*)

Here, *eventHandler* specifies the handler that will be called when a popup request has been received for the context menu. In this case, the handler must call the **show( )** method defined by **ContextMenu** to cause the context menu to be displayed. This is the version we will use:

final void show(Node *node*, double *upperX*, double *upperY*)

Here, *node* is the element on which the context menu is linked. The values of *upperX* and *upperY* define the X,Y location of the upper-left corner of the menu, relative to the screen. Typically, you will pass the screen coordinates at which the right-click occurred. To do this, you will call the **getScreenX( )** and **getScreenY( )** methods defined by **ContextMenuEvent**. They are shown here:

final double getScreenX( )

final double getScreenY( )

Thus, you will typically pass the results of these methods to the **show( )** method.

The preceding theory can be put into practice by adding the context menu to the root node of the scene graph. After doing so, right-clicking anywhere in the scene will cause the menu to pop up. To do this, first add the following sequence to the **MenuDemo** program:

```
// Add the context menu to the entire scene graph.
rootNode.setOnContextMenuRequested(
            new EventHandler<ContextMenuEvent>() {
  public void handle(ContextMenuEvent ae) {
    // Popup menu at the location of the right click.
    editMenu.show(rootNode, ae.getScreenX(), ae.getScreenY());
  }
});
```

Second, declare **rootNode final** so that it can be accessed within the anonymous inner class. After you have made these additions and changes, the context menu can be activated by clicking the right mouse button anywhere inside the application scene. For example, here is the menu displayed after right-clicking in the upper-left portion of the window.

Create a Toolbar

A toolbar is a component that can serve as both an alternative and as an adjunct to a menu. Typically, a toolbar contains a list of buttons that give the user immediate access to various program options. For example, a toolbar might contain buttons that select various font options, such as bold, italics, highlight, or underline. These options can be selected without the need to drop through a menu. As a general rule, toolbar buttons show images rather than text, although either or both are allowed. Furthermore, often tooltips are associated with image-based toolbar buttons.

In JavaFX, toolbars are instances of the **ToolBar** class. It defines the two constructors, shown here:

ToolBar( )

ToolBar(Node ... *nodes*)

The first constructor creates an empty, horizontal toolbar. The second creates a horizontal toolbar that contains the specified nodes, which are usually some form of button. If you want to create a vertical toolbar, call **setOrientation( )** on the toolbar. It is shown here:

final void setOrientation(Orientation *how*)

The value of *how* must be either **Orientation.VERTICAL** or **Orientation.HORIZONTAL**.

You add buttons (or other controls) to a toolbar in much the same way that you add them to a menu bar: call **add( )** on the reference returned by the **getItems( )** method. Often, however, it is easier to specify the items in the **ToolBar** constructor, and that is the approach used in this chapter. Once you have created a toolbar, add it to the scene graph. For example, when using a border layout, it could be added to the bottom location. Of course, other approaches are commonly used. For example, it could be added to a location directly under the menu bar or at the side of the window.

To illustrate a toolbar, we will add one to the **MenuDemo** program. The toolbar will present three debugging options: Set Breakpoint, Clear Breakpoint, and Resume Execution. We will also add tooltips to the menu items. Recall from the previous chapter, a tooltip is a small message that describes an item. It is automatically displayed if the mouse hovers over the item for moment. You can add a tooltip to the menu item in the same way as you add it to a control: by calling **setTooltip( )**. Tooltips are especially useful when applied to image-based toolbar controls because sometimes it's hard to design images that are intuitive to all users.

First, add the following code, which creates the debugging toolbar:

```
// Define a toolbar. First, create toolbar items.
Button btnSet = new Button("Set Breakpoint",
                           new ImageView("setBP.gif"));
Button btnClear = new Button("Clear Breakpoint",
                             new ImageView("clearBP.gif"));
Button btnResume = new Button("Resume Execution",
                              new ImageView("resume.gif"));

// Now, turn off text in the buttons.
btnSet.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);
btnClear.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);
btnResume.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);
```

```
// Set tooltips.
btnSet.setTooltip(new Tooltip("Set a breakpoint."));
btnClear.setTooltip(new Tooltip("Clear a breakpoint."));
btnResume.setTooltip(new Tooltip("Resume execution."));

// Create the toolbar.
ToolBar tbDebug = new ToolBar(btnSet, btnClear, btnResume);
```

Let's look at this code closely. First, three buttons are created that correspond to the debug actions. Notice that each has an image associated with it. Next, each button deactivates the text display by calling **setContentDisplay( )**. As a point of interest, it would have been possible to leave the text displayed, but the toolbar would have had a somewhat nonstandard look. (The text for each button is still needed, however, because it will be used by the action event handler for the buttons.) Tooltips are then set for each button. Finally, the toolbar is created, with the buttons specified as the contents.

Next, add the following sequence, which defines an action event handler for the toolbar buttons:

```
// Create a handler for the toolbar buttons.
EventHandler<ActionEvent> btnHandler = new EventHandler<ActionEvent>() {
  public void handle(ActionEvent ae) {
    response.setText(((Button)ae.getTarget()).getText());
  }
};

// Set the toolbar button action event handlers.
btnSet.setOnAction(btnHandler);
btnClear.setOnAction(btnHandler);
btnResume.setOnAction(btnHandler);
```

Finally, add the toolbar to the bottom of the border layout by using this statement:

```
rootNode.setBottom(tbDebug);
```

After making these additions, each time the user presses a toolbar button, an action event is fired, and it is handled by displaying the button's text in the **response** label. The following output shows the toolbar in action.

Put the Entire MenuDemo Program Together

Throughout the course of this discussion, many changes and additions have been made to the **MenuDemo** program shown at the start of the chapter. Before concluding, it will be helpful to assemble all the pieces. Doing so not only eliminates any ambiguity about the way the pieces fit together, but it also gives you a complete menu demonstration program that you can experiment with.

The following version of **MenuDemo** includes all of the additions and enhancements described in this chapter. For clarity, the program has been reorganized, with separate methods being used to construct the various menus and toolbar. Notice that several of the menu-related variables, such as **mb** and **tbDebug**, have been made into instance variables so they can be directly accessed by any part of the class.

```
// Demonstrate Menus -- Final Version

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.input.*;
import javafx.scene.image.*;
import javafx.beans.value.*;

public class MenuDemoFinal extends Application {

  MenuBar mb;
  EventHandler<ActionEvent> MEHandler;
  ContextMenu editMenu;
  ToolBar tbDebug;

  Label response;

  public static void main(String[] args) {

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the start() method.
  public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("Demonstrate Menus -- Final Version");

    // Use a BorderPane for the root node.
    final BorderPane rootNode = new BorderPane();

     // Create a scene.
    Scene myScene = new Scene(rootNode, 300, 300);
```

```
// Set the scene on the stage.
myStage.setScene(myScene);

// Create a label that will report the selection.
response = new Label();

// Create one event handler for all menu action events.
MEHandler = new EventHandler<ActionEvent>() {
  public void handle(ActionEvent ae) {
    String name = ((MenuItem)ae.getTarget()).getText();

    if(name.equals("Exit")) Platform.exit();

    response.setText( name + " selected");
  }
};

// Create the menu bar.
mb = new MenuBar();

// Create the File menu.
makeFileMenu();

// Create the Options menu.
makeOptionsMenu();

// Create the Help menu.
makeHelpMenu();

// Create the context menu.
makeContextMenu();

// Create a text field and set its column width to 20.
TextField tf = new TextField();
tf.setPrefColumnCount(20);

// Add the context menu to the text field.
tf.setContextMenu(editMenu);

// Create the toolbar.
makeToolBar();

// Add the context menu to the entire scene graph.
rootNode.setOnContextMenuRequested(
            new EventHandler<ContextMenuEvent>() {
  public void handle(ContextMenuEvent ae) {
    // Popup menu at the location of the right click.
    editMenu.show(rootNode, ae.getScreenX(), ae.getScreenY());
  }
});

// Add the menu bar to the top of the border pane.
rootNode.setTop(mb);
```

```
    // Create a flow pane that will hold both the response
    // label and the text field.
    FlowPane fpRoot = new FlowPane(10, 10);

    // Center the controls in the scene.
    fpRoot.setAlignment(Pos.CENTER);

    // Use a separator to better organize the layout.
    Separator separator = new Separator();
    separator.setPrefWidth(260);

    // Add the label, separator, and text field to the flow pane.
    fpRoot.getChildren().addAll(response, separator,  tf);

    // Add the toolbar to the bottom of the border pane.
    rootNode.setBottom(tbDebug);

    // Add the flow pane to the center of the border layout.
    rootNode.setCenter(fpRoot);

    // Show the stage and its scene.
    myStage.show();
  }

  // Create the File menu.
  void makeFileMenu() {
    // Create the File menu, including a mnemonic.
    Menu fileMenu = new Menu("_File");

    // Create the File menu items.
    MenuItem open = new MenuItem("Open");
    MenuItem close = new MenuItem("Close");
    MenuItem save = new MenuItem("Save");
    MenuItem exit = new MenuItem("Exit");

    // Add items to File menu.
    fileMenu.getItems().addAll(open, close, save,
                               new SeparatorMenuItem(), exit);

    // Add keyboard accelerators for the File menu.
    open.setAccelerator(KeyCombination.keyCombination("shortcut+O"));
    close.setAccelerator(KeyCombination.keyCombination("shortcut+C"));
    save.setAccelerator(KeyCombination.keyCombination("shortcut+S"));
    exit.setAccelerator(KeyCombination.keyCombination("shortcut+E"));

    // Set action event handlers.
    open.setOnAction(MEHandler);
    close.setOnAction(MEHandler);
    save.setOnAction(MEHandler);
    exit.setOnAction(MEHandler);

    // Add File menu to the menu bar.
    mb.getMenus().add(fileMenu);
  }
```

```
// Create the Options menu.
void makeOptionsMenu() {
  Menu optionsMenu = new Menu("Options");

  // Create the Colors submenu.
  Menu colorsMenu = new Menu("Colors");

  // Use check menu items for colors. This allows
  // the user to select more than one color.
  CheckMenuItem red = new CheckMenuItem("Red");
  CheckMenuItem green = new CheckMenuItem("Green");
  CheckMenuItem blue = new CheckMenuItem("Blue");

  // Add the check menu items for the Colors menu and
  // add the colors menu to the Options menu.
  colorsMenu.getItems().addAll(red, green, blue);
  optionsMenu.getItems().add(colorsMenu);

  // Select green for the default color selection.
  green.setSelected(true);

  // Create the Priority submenu.
  Menu priorityMenu = new Menu("Priority");

  // Use radio menu items for the priority setting.
  // This lets the menu show which priority is used
  // and also ensures that one and only one priority
  // can be selected at any one time.
  RadioMenuItem high = new RadioMenuItem("High");
  RadioMenuItem low = new RadioMenuItem("Low");

  // Create a toggle group and use it for the radio menu items.
  ToggleGroup tg = new ToggleGroup();
  high.setToggleGroup(tg);
  low.setToggleGroup(tg);

  // Select High priority for the default selection.
  high.setSelected(true);

  // Add the radio menu items to the Priority menu and
  // add the Priority menu to the Options menu.
  priorityMenu.getItems().addAll(high, low);
  optionsMenu.getItems().add(priorityMenu);

  // Add a separator.
  optionsMenu.getItems().add(new SeparatorMenuItem());

  // Create the Reset menu item and add it to the Options menu.
  MenuItem reset = new MenuItem("Reset");
  optionsMenu.getItems().add(reset);

  // Set action event handlers.
  red.setOnAction(MEHandler);
  green.setOnAction(MEHandler);
  blue.setOnAction(MEHandler);
```

```
    high.setOnAction(MEHandler);
    low.setOnAction(MEHandler);
    reset.setOnAction(MEHandler);

    // Use a change listener to respond to changes in the radio
    // menu item setting.
    tg.selectedToggleProperty().addListener(new ChangeListener<Toggle>() {
      public void changed(ObservableValue<? extends Toggle> changed,
                          Toggle oldVal, Toggle newVal) {
        if(newVal==null) return;

        // Cast newVal to RadioButton.
        RadioMenuItem rmi = (RadioMenuItem) newVal;

        // Display the selection.
        response.setText("Priority selected is " + rmi.getText());
      }
    });

    // Add Options menu to the menu bar.
    mb.getMenus().add(optionsMenu);
  }

  // Create the Help menu.
  void makeHelpMenu() {

    // Create an ImageView for the image.
    ImageView aboutIV = new ImageView("aboutIcon.gif");

    // Create the Help menu.
    Menu helpMenu = new Menu("Help");

    // Create the About menu item and add it to the Help menu.
    MenuItem about = new MenuItem("About", aboutIV);
    helpMenu.getItems().add(about);

    // Set action event handler.
    about.setOnAction(MEHandler);

    // Add Help menu to the menu bar.
    mb.getMenus().add(helpMenu);
  }

  // Create the context menu items.
  void makeContextMenu() {

    // Create the edit context menu items.
    MenuItem cut = new MenuItem("Cut");
    MenuItem copy = new MenuItem("Copy");
    MenuItem paste = new MenuItem("Paste");

    // Create a context (i.e., popup) menu that shows edit options.
    editMenu = new ContextMenu(cut, copy, paste);
```

```
    // Set the action event handlers.
    cut.setOnAction(MEHandler);
    copy.setOnAction(MEHandler);
    paste.setOnAction(MEHandler);
  }

  // Create the toolbar.
  void makeToolBar() {
    // Create toolbar items.
    Button btnSet = new Button("Set Breakpoint",
                               new ImageView("setBP.gif"));
    Button btnClear = new Button("Clear Breakpoint",
                                 new ImageView("clearBP.gif"));
    Button btnResume = new Button("Resume Execution",
                                  new ImageView("resume.gif"));

    // Turn off text in the buttons.
    btnSet.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);
    btnClear.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);
    btnResume.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);

    // Set tooltips.
    btnSet.setTooltip(new Tooltip("Set a breakpoint."));
    btnClear.setTooltip(new Tooltip("Clear a breakpoint."));
    btnResume.setTooltip(new Tooltip("Resume execution."));

    // Create the toolbar.
    tbDebug = new ToolBar(btnSet, btnClear, btnResume);

    // Create a handler for the toolbar buttons.
    EventHandler<ActionEvent> btnHandler = new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        response.setText(((Button)ae.getTarget()).getText());
      }
    };

    // Set the toolbar button action event handlers.
    btnSet.setOnAction(btnHandler);
    btnClear.setOnAction(btnHandler);
    btnResume.setOnAction(btnHandler);
  }
}
```

Continuing Your Exploration of JavaFX

JavaFX represents a major advance in GUI frameworks for Java. It also redefines aspects of the Java platform. The preceding three chapters have introduced several of its core features, but there is much left to explore. For example, JavaFX supplies several more controls, such as sliders, stand-alone scrollbars, and tables. You will want to experiment with its layouts, such as **VBox** and **Hbox**. You will also want to explore, in detail, the various effects in **javafx.scene.effect** and the various transforms in **javafx.scene.transform**. Another exciting class is **WebView**, which gives you an easy way to integrate web content into a scene graph. Frankly, all of JavaFX is worthy of serious study. In many ways, it is charting the future course of Java.

PART

V

Applying Java

This page has been intentionally left blank

CHAPTER

37 Java Beans

This chapter provides an overview of Java Beans. Beans are important because they allow you to build complex systems from software components. These components may be provided by you or supplied by one or more different vendors. Java Beans defines an architecture that specifies how these building blocks can operate together.

To better understand the value of Beans, consider the following. Hardware designers have a wide variety of components that can be integrated together to construct a system. Resistors, capacitors, and inductors are examples of simple building blocks. Integrated circuits provide more advanced functionality. All of these different parts can be reused. It is not necessary or possible to rebuild these capabilities each time a new system is needed. Also, the same pieces can be used in different types of circuits. This is possible because the behavior of these components is understood and documented.

The software industry has also been seeking the benefits of reusability and interoperability of a component-based approach. To realize these benefits, a component architecture is needed that allows programs to be assembled from software building blocks, perhaps provided by different vendors. It must also be possible for a designer to select a component, understand its capabilities, and incorporate it into an application. When a new version of a component becomes available, it should be easy to incorporate this functionality into existing code. Fortunately, Java Beans provides just such an architecture.

What Is a Java Bean?

A *Java Bean* is a software component that has been designed to be reusable in a variety of different environments. There is no restriction on the capability of a Bean. It may perform a simple function, such as obtaining an inventory value, or a complex function, such as forecasting the performance of a stock portfolio. A Bean may be visible to an end user. One example of this is a button on a graphical user interface. A Bean may also be invisible to a user. Software to decode a stream of multimedia information in real time is an example of this type of building block. Finally, a Bean may be designed to work autonomously on a user's workstation or to work in cooperation with a set of other distributed components.

Software to generate a pie chart from a set of data points is an example of a Bean that can execute locally. However, a Bean that provides real-time price information from a stock or commodities exchange would need to work in cooperation with other distributed software to obtain its data.

Advantages of Java Beans

The following list enumerates some of the benefits that Java Bean technology provides for a component developer:

- A Bean obtains all the benefits of Java's "write-once, run-anywhere" paradigm.
- The properties, events, and methods of a Bean that are exposed to another application can be controlled.
- Auxiliary software can be provided to help configure a Bean. This software is only needed when the design-time parameters for that component are being set. It does not need to be included in the run-time environment.
- The state of a Bean can be saved in persistent storage and restored at a later time.
- A Bean may register to receive events from other objects and can generate events that are sent to other objects.

Introspection

At the core of Java Beans is *introspection*. This is the process of analyzing a Bean to determine its capabilities. This is an essential feature of the Java Beans API because it allows another application, such as a design tool, to obtain information about a component. Without introspection, the Java Beans technology could not operate.

There are two ways in which the developer of a Bean can indicate which of its properties, events, and methods should be exposed. With the first method, simple naming conventions are used. These allow the introspection mechanisms to infer information about a Bean. In the second way, an additional class that extends the **BeanInfo** interface is provided that explicitly supplies this information. Both approaches are examined here.

Design Patterns for Properties

A *property* is a subset of a Bean's state. The values assigned to the properties determine the behavior and appearance of that component. A property is set through a *setter* method. A property is obtained by a *getter* method. There are two types of properties: simple and indexed.

Simple Properties

A simple property has a single value. It can be identified by the following design patterns, where **N** is the name of the property and **T** is its type:

```
public T getN( )
public void setN(T arg)
```

A read/write property has both of these methods to access its values. A read-only property has only a get method. A write-only property has only a set method.

Here are three read/write simple properties along with their getter and setter methods:

```
private double depth, height, width;

public double getDepth( ) {
  return depth;
}
public void setDepth(double d) {
  depth = d;
}

public double getHeight( ) {
  return height;
}
public void setHeight(double h) {
  height = h;
}

public double getWidth( ) {
  return width;
}
public void setWidth(double w) {
  width = w;
}
```

NOTE For a **boolean** property, a method of the form **is***PropertyName*( ) can also be used as an accessor.

Indexed Properties

An indexed property consists of multiple values. It can be identified by the following design patterns, where **N** is the name of the property and **T** is its type:

public T getN(int *index*);
public void setN(int *index*, T *value*);
public T[ ] getN( );
public void setN(T *values*[ ]);

Here is an indexed property called **data** along with its getter and setter methods:

```
private double data[ ];

public double getData(int index) {
  return data[index];
}
public void setData(int index, double value) {
  data[index] = value;
}
public double[ ] getData( ) {
  return data;
}
public void setData(double[ ] values) {
  data = new double[values.length];
  System.arraycopy(values, 0, data, 0, values.length);
}
```

Design Patterns for Events

Beans use the delegation event model that was discussed earlier in this book. Beans can generate events and send them to other objects. These can be identified by the following design patterns, where **T** is the type of the event:

public void addTListener(TListener *eventListener*)
public void addTListener(TListener *eventListener*)
throws java.util.TooManyListenersException
public void removeTListener(TListener *eventListener*)

These methods are used to add or remove a listener for the specified event. The version of **addTListener( )** that does not throw an exception can be used to *multicast* an event, which means that more than one listener can register for the event notification. The version that throws **TooManyListenersException** *unicasts* the event, which means that the number of listeners can be restricted to one. In either case, **removeTListener( )** is used to remove the listener. For example, assuming an event interface type called **TemperatureListener**, a Bean that monitors temperature might supply the following methods:

```
public void addTemperatureListener(TemperatureListener tl) {
  ...
}
public void removeTemperatureListener(TemperatureListener tl) {
  ...
}
```

Methods and Design Patterns

Design patterns are not used for naming nonproperty methods. The introspection mechanism finds all of the public methods of a Bean. Protected and private methods are not presented.

Using the BeanInfo Interface

As the preceding discussion shows, design patterns *implicitly* determine what information is available to the user of a Bean. The **BeanInfo** interface enables you to *explicitly* control what information is available. The **BeanInfo** interface defines several methods, including these:

PropertyDescriptor[ ] getPropertyDescriptors( )
EventSetDescriptor[ ] getEventSetDescriptors( )
MethodDescriptor[ ] getMethodDescriptors( )

They return arrays of objects that provide information about the properties, events, and methods of a Bean. The classes **PropertyDescriptor**, **EventSetDescriptor**, and **MethodDescriptor** are defined within the **java.beans** package, and they describe the indicated elements. By implementing these methods, a developer can designate exactly what is presented to a user, bypassing introspection based on design patterns.

When creating a class that implements **BeanInfo**, you must call that class *bname*BeanInfo, where *bname* is the name of the Bean. For example, if the Bean is called **MyBean**, then the information class must be called **MyBeanBeanInfo**.

To simplify the use of **BeanInfo**, JavaBeans supplies the **SimpleBeanInfo** class. It provides default implementations of the **BeanInfo** interface, including the three methods just shown. You can extend this class and override one or more of the methods to explicitly control what aspects of a Bean are exposed. If you don't override a method, then design-pattern introspection will be used. For example, if you don't override **getPropertyDescriptors( )**, then design patterns are used to discover a Bean's properties. You will see **SimpleBeanInfo** in action later in this chapter.

Bound and Constrained Properties

A Bean that has a *bound* property generates an event when the property is changed. The event is of type **PropertyChangeEvent** and is sent to objects that previously registered an interest in receiving such notifications. A class that handles this event must implement the **PropertyChangeListener** interface.

A Bean that has a *constrained* property generates an event when an attempt is made to change its value. It also generates an event of type **PropertyChangeEvent**. It too is sent to objects that previously registered an interest in receiving such notifications. However, those other objects have the ability to veto the proposed change by throwing a **PropertyVetoException**. This capability allows a Bean to operate differently according to its run-time environment. A class that handles this event must implement the **VetoableChangeListener** interface.

Persistence

Persistence is the ability to save the current state of a Bean, including the values of a Bean's properties and instance variables, to nonvolatile storage and to retrieve them at a later time. The object serialization capabilities provided by the Java class libraries are used to provide persistence for Beans.

The easiest way to serialize a Bean is to have it implement the **java.io.Serializable** interface, which is simply a marker interface. Implementing **java.io.Serializable** makes serialization automatic. Your Bean need take no other action. Automatic serialization can also be inherited. Therefore, if any superclass of a Bean implements **java.io.Serializable**, then automatic serialization is obtained.

When using automatic serialization, you can selectively prevent a field from being saved through the use of the **transient** keyword. Thus, data members of a Bean specified as **transient** will not be serialized.

If a Bean does not implement **java.io.Serializable**, you must provide serialization yourself, such as by implementing **java.io.Externalizable**. Otherwise, containers cannot save the configuration of your component.

Customizers

A Bean developer can provide a *customizer* that helps another developer configure the Bean. A customizer can provide a step-by-step guide through the process that must be followed to use the component in a specific context. Online documentation can also be provided. A Bean developer has great flexibility to develop a customizer that can differentiate his or her product in the marketplace.

Interface	Description
AppletInitializer	Methods in this interface are used to initialize Beans that are also applets.
BeanInfo	This interface allows a designer to specify information about the properties, events, and methods of a Bean.
Customizer	This interface allows a designer to provide a graphical user interface through which a Bean may be configured.
DesignMode	Methods in this interface determine if a Bean is executing in design mode.
ExceptionListener	A method in this interface is invoked when an exception has occurred.
PropertyChangeListener	A method in this interface is invoked when a bound property is changed.
PropertyEditor	Objects that implement this interface allow designers to change and display property values.
VetoableChangeListener	A method in this interface is invoked when a constrained property is changed.
Visibility	Methods in this interface allow a Bean to execute in environments where a graphical user interface is not available.

Table 37-1 The Interfaces in **java.beans**

The Java Beans API

The Java Beans functionality is provided by a set of classes and interfaces in the **java.beans** package. This section provides a brief overview of its contents. Table 37-1 lists the interfaces in **java.beans** and provides a brief description of their functionality. Table 37-2 lists the classes in **java.beans**.

Class	Description
BeanDescriptor	This class provides information about a Bean. It also allows you to associate a customizer with a Bean.
Beans	This class is used to obtain information about a Bean.
DefaultPersistenceDelegate	A concrete subclass of **PersistenceDelegate**.
Encoder	Encodes the state of a set of Beans. Can be used to write this information to a stream.
EventHandler	Supports dynamic event listener creation.
EventSetDescriptor	Instances of this class describe an event that can be generated by a Bean.
Expression	Encapsulates a call to a method that returns a result.
FeatureDescriptor	This is the superclass of the **PropertyDescriptor**, **EventSetDescriptor**, and **MethodDescriptor** classes, among others.

Table 37-2 The Classes in **java.beans**

Class	Description
IndexedPropertyChangeEvent	A subclass of **PropertyChangeEvent** that represents a change to an indexed property.
IndexedPropertyDescriptor	Instances of this class describe an indexed property of a Bean.
IntrospectionException	An exception of this type is generated if a problem occurs when analyzing a Bean.
Introspector	This class analyzes a Bean and constructs a **BeanInfo** object that describes the component.
MethodDescriptor	Instances of this class describe a method of a Bean.
ParameterDescriptor	Instances of this class describe a method parameter.
PersistenceDelegate	Handles the state information of an object.
PropertyChangeEvent	This event is generated when bound or constrained properties are changed. It is sent to objects that registered an interest in these events and that implement either the **PropertyChangeListener** or **VetoableChangeListener** interfaces.
PropertyChangeListenerProxy	Extends **EventListenerProxy** and implements **PropertyChangeListener**.
PropertyChangeSupport	Beans that support bound properties can use this class to notify **PropertyChangeListener** objects.
PropertyDescriptor	Instances of this class describe a property of a Bean.
PropertyEditorManager	This class locates a **PropertyEditor** object for a given type.
PropertyEditorSupport	This class provides functionality that can be used when writing property editors.
PropertyVetoException	An exception of this type is generated if a change to a constrained property is vetoed.
SimpleBeanInfo	This class provides functionality that can be used when writing **BeanInfo** classes.
Statement	Encapsulates a call to a method.
VetoableChangeListenerProxy	Extends **EventListenerProxy** and implements **VetoableChangeListener**.
VetoableChangeSupport	Beans that support constrained properties can use this class to notify **VetoableChangeListener** objects.
XMLDecoder	Used to read a Bean from an XML document.
XMLEncoder	Used to write a Bean to an XML document.

Table 37-2 The Classes in **java.beans** *(continued)*

Although it is beyond the scope of this chapter to discuss all of the classes, four are of particular interest: **Introspector**, **PropertyDescriptor**, **EventSetDescriptor**, and **MethodDescriptor**. Each is briefly examined here.

Introspector

The **Introspector** class provides several static methods that support introspection. Of most interest is **getBeanInfo( )**. This method returns a **BeanInfo** object that can be used to obtain information about the Bean. The **getBeanInfo( )** method has several forms, including the one shown here:

static BeanInfo getBeanInfo(Class<?> *bean*) throws IntrospectionException

The returned object contains information about the Bean specified by *bean*.

PropertyDescriptor

The **PropertyDescriptor** class describes the characteristics of a Bean property. It supports several methods that manage and describe properties. For example, you can determine if a property is bound by calling **isBound( )**. To determine if a property is constrained, call **isConstrained( )**. You can obtain the name of a property by calling **getName( )**.

EventSetDescriptor

The **EventSetDescriptor** class represents a Bean event. It supports several methods that obtain the methods that a Bean uses to add or remove event listeners, and to otherwise manage events. For example, to obtain the method used to add listeners, call **getAddListenerMethod( )**. To obtain the method used to remove listeners, call **getRemoveListenerMethod( )**. To obtain the type of a listener, call **getListenerType( )**. You can obtain the name of an event by calling **getName( )**.

MethodDescriptor

The **MethodDescriptor** class represents a Bean method. To obtain the name of the method, call **getName( )**. You can obtain information about the method by calling **getMethod( )**, shown here:

Method getMethod( )

An object of type **Method** that describes the method is returned.

A Bean Example

This chapter concludes with an example that illustrates various aspects of Bean programming, including introspection and using a **BeanInfo** class. It also makes use of the **Introspector**, **PropertyDescriptor**, and **EventSetDescriptor** classes. The example uses three classes. The first is a Bean called **Colors**, shown here:

```
// A simple Bean.
import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;

public class Colors extends Canvas implements Serializable {
  transient private Color color; // not persistent
  private boolean rectangular; // is persistent
```

```
  public Colors() {
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent me) {
        change();
      }
    });
    rectangular = false;
    setSize(200, 100);
    change();
  }

  public boolean getRectangular() {
    return rectangular;
  }

  public void setRectangular(boolean flag) {
    this.rectangular = flag;
    repaint();
  }

  public void change() {
    color = randomColor();
    repaint();
  }

  private Color randomColor() {
    int r = (int)(255*Math.random());
    int g = (int)(255*Math.random());
    int b = (int)(255*Math.random());
    return new Color(r, g, b);
  }

  public void paint(Graphics g) {
    Dimension d = getSize();
    int h = d.height;
    int w = d.width;
    g.setColor(color);
    if(rectangular) {
      g.fillRect(0, 0, w-1, h-1);
    }
    else {
      g.fillOval(0, 0, w-1, h-1);
    }
  }
}
```

The **Colors** Bean displays a colored object within a frame. The color of the component is determined by the private **Color** variable **color**, and its shape is determined by the private **boolean** variable **rectangular**. The constructor defines an anonymous inner class that extends **MouseAdapter** and overrides its **mousePressed( )** method. The **change( )** method is invoked in response to mouse presses. It selects a random color and then repaints the component. The **getRectangular( )** and **setRectangular( )** methods provide access to the one property

of this Bean. The **change( )** method calls **randomColor( )** to choose a color and then calls **repaint( )** to make the change visible. Notice that the **paint( )** method uses the **rectangular** and **color** variables to determine how to present the Bean.

The next class is **ColorsBeanInfo**. It is a subclass of **SimpleBeanInfo** that provides explicit information about **Colors**. It overrides **getPropertyDescriptors( )** in order to designate which properties are presented to a Bean user. In this case, the only property exposed is **rectangular**. The method creates and returns a **PropertyDescriptor** object for the **rectangular** property. The **PropertyDescriptor** constructor that is used is shown here:

PropertyDescriptor(String *property*, Class<?> *beanCls*)
  throws IntrospectionException

Here, the first argument is the name of the property, and the second argument is the class of the Bean.

```
// A Bean information class.
import java.beans.*;
public class ColorsBeanInfo extends SimpleBeanInfo {
  public PropertyDescriptor[] getPropertyDescriptors() {
    try {
      PropertyDescriptor rectangular = new
         PropertyDescriptor("rectangular", Colors.class);
      PropertyDescriptor pd[] = {rectangular};
      return pd;
    }
    catch(Exception e) {
      System.out.println("Exception caught. " + e);
    }
    return null;
  }
}
```

The final class is called **IntrospectorDemo**. It uses introspection to display the properties and events that are available within the **Colors** Bean.

```
// Show properties and events.
import java.awt.*;
import java.beans.*;

public class IntrospectorDemo {
  public static void main(String args[]) {
    try {
      Class<?> c = Class.forName("Colors");
      BeanInfo beanInfo = Introspector.getBeanInfo(c);

      System.out.println("Properties:");
      PropertyDescriptor propertyDescriptor[] =
        beanInfo.getPropertyDescriptors();
      for(int i = 0; i < propertyDescriptor.length; i++) {
        System.out.println("\t" + propertyDescriptor[i].getName());
      }
```

```
      System.out.println("Events:");
      EventSetDescriptor eventSetDescriptor[] =
        beanInfo.getEventSetDescriptors();
      for(int i = 0; i < eventSetDescriptor.length; i++) {
        System.out.println("\t" + eventSetDescriptor[i].getName());
      }
    }
    catch(Exception e) {
      System.out.println("Exception caught. " + e);
    }
  }
}
```

The output from this program is the following:

```
Properties:
        rectangular
Events:
        mouseWheel
        mouse
        mouseMotion
        component
        hierarchyBounds
        focus
        hierarchy
        propertyChange
        inputMethod
        key
```

Notice two things in the output. First, because **ColorsBeanInfo** overrides **getPropertyDescriptors( )** such that the only property returned is **rectangular**, only the **rectangular** property is displayed. However, because **getEventSetDescriptors( )** is not overridden by **ColorsBeanInfo**, design-pattern introspection is used, and all events are found, including those in **Colors**' superclass, **Canvas**. Remember, if you don't override one of the "get" methods defined by **SimpleBeanInfo**, then the default, design-pattern introspection is used. To observe the difference that **ColorsBeanInfo** makes, erase its class file and then run **IntrospectorDemo** again. This time it will report more properties.

This page has been intentionally left blank

CHAPTER

38 Introducing Servlets

This chapter presents an introduction to *servlets*. Servlets are small programs that execute on the server side of a web connection. Just as applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server. The topic of servlets is quite large, and it is beyond the scope of this chapter to cover it all. Instead, we will focus on the core concepts, interfaces, and classes, and develop several examples.

Background

In order to understand the advantages of servlets, you must have a basic understanding of how web browsers and servers cooperate to provide content to a user. Consider a request for a static web page. A user enters a Uniform Resource Locator (URL) into a browser. The browser generates an HTTP request to the appropriate web server. The web server maps this request to a specific file. That file is returned in an HTTP response to the browser. The HTTP header in the response indicates the type of the content. The Multipurpose Internet Mail Extensions (MIME) are used for this purpose. For example, ordinary ASCII text has a MIME type of text/plain. The Hypertext Markup Language (HTML) source code of a web page has a MIME type of text/html.

Now consider dynamic content. Assume that an online store uses a database to store information about its business. This would include items for sale, prices, availability, orders, and so forth. It wishes to make this information accessible to customers via web pages. The contents of those web pages must be dynamically generated to reflect the latest information in the database.

In the early days of the Web, a server could dynamically construct a page by creating a separate process to handle each client request. The process would open connections to one or more databases in order to obtain the necessary information. It communicated with the web server via an interface known as the Common Gateway Interface (CGI). CGI allowed the separate process to read data from the HTTP request and write data to the HTTP response. A variety of different languages were used to build CGI programs. These included C, C++, and Perl.

However, CGI suffered serious performance problems. It was expensive in terms of processor and memory resources to create a separate process for each client request. It was also expensive to open and close database connections for each client request. In addition, the CGI programs were not platform-independent. Therefore, other techniques were introduced. Among these are servlets.

Servlets offer several advantages in comparison with CGI. First, performance is significantly better. Servlets execute within the address space of a web server. It is not necessary to create a separate process to handle each client request. Second, servlets are platform-independent because they are written in Java. Third, the Java security manager on the server enforces a set of restrictions to protect the resources on a server machine. Finally, the full functionality of the Java class libraries is available to a servlet. It can communicate with applets, databases, or other software via the sockets and RMI mechanisms that you have seen already.

The Life Cycle of a Servlet

Three methods are central to the life cycle of a servlet. These are **init( )**, **service( )**, and **destroy( )**. They are implemented by every servlet and are invoked at specific times by the server. Let us consider a typical user scenario to understand when these methods are called.

First, assume that a user enters a Uniform Resource Locator (URL) to a web browser. The browser then generates an HTTP request for this URL. This request is then sent to the appropriate server.

Second, this HTTP request is received by the web server. The server maps this request to a particular servlet. The servlet is dynamically retrieved and loaded into the address space of the server.

Third, the server invokes the **init( )** method of the servlet. This method is invoked only when the servlet is first loaded into memory. It is possible to pass initialization parameters to the servlet so it may configure itself.

Fourth, the server invokes the **service( )** method of the servlet. This method is called to process the HTTP request. You will see that it is possible for the servlet to read data that has been provided in the HTTP request. It may also formulate an HTTP response for the client.

The servlet remains in the server's address space and is available to process any other HTTP requests received from clients. The **service( )** method is called for each HTTP request.

Finally, the server may decide to unload the servlet from its memory. The algorithms by which this determination is made are specific to each server. The server calls the **destroy( )** method to relinquish any resources such as file handles that are allocated for the servlet. Important data may be saved to a persistent store. The memory allocated for the servlet and its objects can then be garbage collected.

Servlet Development Options

To create servlets, you will need access to a servlet container/server. Two popular ones are Glassfish and Tomcat. Glassfish is from Oracle and is provided by the Java EE SDK. It is supported by NetBeans. Tomcat is an open-source product maintained by the Apache Software Foundation. It can also be used by NetBeans. Both Tomcat and Glassfish can also be used with other IDEs, such as Eclipse. The examples and descriptions in this chapter use Tomcat for reasons that will soon be apparent.

Although IDEs such as NetBeans and Eclipse are very useful and can streamline the creation of servlets, they are not used in this chapter. The way you develop and deploy servlets differs among IDEs, and it is simply not possible for this book to address each environment. Furthermore, many readers will be using the command-line tools rather than an IDE. Therefore, if you are using an IDE, you must refer to the instructions for that environment for information concerning the development and deployment of servlets. For this reason, the instructions given here and elsewhere in this chapter assume that only the command-line tools are employed. Thus, they will work for nearly any reader.

Tomcat is used in this chapter because, in the opinion of this author, it makes it relatively easy to run the example servlets using only command-line tools and a text editor. It is also widely available in various programming environments. Furthermore, since only command-line tools are used, you don't need to download and install an IDE just to experiment with servlets. Understand, however, that even if you are developing in an environment that uses Glassfish, the concepts presented here still apply. It is just that the mechanics of preparing a servlet for testing will be slightly different.

REMEMBER The instructions for developing and deploying servlets in this chapter are based on Tomcat and use only command-line tools. If you are using an IDE and different servlet container/server, consult the documentation for your environment.

Using Tomcat

Tomcat contains the class libraries, documentation, and run-time support that you will need to create and test servlets. At the time of this writing, several versions of Tomcat are available. The instructions that follow use 7.0.47. You can download Tomcat from **tomcat.apache.org**. You should choose a version appropriate to your environment.

The examples in this chapter assume a 64-bit Windows environment. Assuming that a 64-bit version of Tomcat 7.0.47 was unpacked from the root directly, the default location is

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\
```

This is the location assumed by the examples in this book. If you load Tomcat in a different location (or use a different version of Tomcat), you will need to make appropriate changes to the examples. You may need to set the environmental variable **JAVA_HOME** to the top-level directory in which the Java Development Kit is installed.

NOTE All of the directories shown in this section assume Tomcat 7.0.47. If you install a different version of Tomcat, then you will need to adjust the directory names and paths to match those used by the version you installed.

Once installed, you start Tomcat by selecting **startup.bat** from the **bin** directly under the **apache-tomcat-7.0.47** directory. To stop Tomcat, execute **shutdown.bat**, also in the **bin** directory.

The classes and interfaces needed to build servlets are contained in **servlet-api.jar**, which is in the following directory:

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\lib
```

To make **servlet-api.jar** accessible, update your **CLASSPATH** environment variable so that it includes

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\lib\servlet-api.jar
```

Alternatively, you can specify this file when you compile the servlets. For example, the following command compiles the first servlet example:

```
javac HelloServlet.java -classpath "C:\apache-tomcat-7.0.47-windows-
x64\apache-tomcat-7.0.47\lib\servlet-api.jar"
```

Once you have compiled a servlet, you must enable Tomcat to find it. For our purposes, this means putting it into a directory under Tomcat's **webapps** directory and entering its name into a **web.xml** file. To keep things simple, the examples in this chapter use the directory and **web.xml** file that Tomcat supplies for its own example servlets. This way, you won't have to create any files or directories just to experiment with the sample servlets. Here is the procedure that you will follow.

First, copy the servlet's class file into the following directory:

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\webapps\
examples\WEB-INF\classes
```

Next, add the servlet's name and mapping to the **web.xml** file in the following directory:

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\webapps\
examples\WEB-INF
```

For instance, assuming the first example, called **HelloServlet**, you will add the following lines in the section that defines the servlets:

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>HelloServlet</servlet-class>
</servlet>
```

Next, you will add the following lines to the section that defines the servlet mappings:

```
<servlet-mapping>
  <servlet-name>HelloServlet</servlet-name>
  <url-pattern>/servlet/HelloServlet</url-pattern>
</servlet-mapping>
```

Follow this same general procedure for all of the examples.

A Simple Servlet

To become familiar with the key servlet concepts, we will begin by building and testing a simple servlet. The basic steps are the following:

1. Create and compile the servlet source code. Then, copy the servlet's class file to the proper directory, and add the servlet's name and mappings to the proper **web.xml** file.

2. Start Tomcat.
3. Start a web browser and request the servlet.

Let us examine each of these steps in detail.

Create and Compile the Servlet Source Code

To begin, create a file named **HelloServlet.java** that contains the following program:

```
import java.io.*;
import javax.servlet.*;

public class HelloServlet extends GenericServlet {

  public void service(ServletRequest request,
    ServletResponse response)
  throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>Hello!");
    pw.close();
  }
}
```

Let's look closely at this program. First, note that it imports the **javax.servlet** package. This package contains the classes and interfaces required to build servlets. You will learn more about these later in this chapter. Next, the program defines **HelloServlet** as a subclass of **GenericServlet**. The **GenericServlet** class provides functionality that simplifies the creation of a servlet. For example, it provides versions of **init( )** and **destroy( )**, which may be used as is. You need supply only the **service( )** method.

Inside **HelloServlet**, the **service( )** method (which is inherited from **GenericServlet**) is overridden. This method handles requests from a client. Notice that the first argument is a **ServletRequest** object. This enables the servlet to read data that is provided via the client request. The second argument is a **ServletResponse** object. This enables the servlet to formulate a response for the client.

The call to **setContentType( )** establishes the MIME type of the HTTP response. In this program, the MIME type is text/html. This indicates that the browser should interpret the content as HTML source code.

Next, the **getWriter( )** method obtains a **PrintWriter**. Anything written to this stream is sent to the client as part of the HTTP response. Then **println( )** is used to write some simple HTML source code as the HTTP response.

Compile this source code and place the **HelloServlet.class** file in the proper Tomcat directory as described in the previous section. Also, add **HelloServlet** to the **web.xml** file, as described earlier.

Start Tomcat

Start Tomcat as explained earlier. Tomcat must be running before you try to execute a servlet.