Chapter 10

Graphical User Interfaces in Java

Users typically interact with computers via *graphical user interfaces* or *GUI*'s. These interfaces provide the user with multiple windows on the screen and support the use of a mouse to click on buttons, drag items around, pull down and select items from menus, select text fields in which to type responses, scroll through windows, and perform many other operations. GUI components (sometimes called "widgets") are items like buttons and menus that can be added to the user interface to provide a way for users to interact with a program.

So far we have not been able to program using these components. We can draw geometric objects on a canvas and interact with programs using mouse actions, but we haven't yet seen how to create and interact with GUI components in the way that one does with most programs running on personal computers. In this chapter we will introduce you to techniques for programming a graphical user interface with your Java programs.

Java provides two libraries to assist in the programming of GUI interfaces, AWT, standing for Abstract Windowing Toolkit, and Swing. Swing is intended to replace AWT, Java's original windowing toolkit. In this chapter we will focus on Swing. However, many aspects of GUI programming still depend on AWT classes, so we will use classes from both of these libraries as well as libraries providing support for event handling.

A thorough discussion of the Swing package and how to use it could fill an entire book. In this chapter we will introduce some of the basic features of the package. More detailed information on the package should be available on your local computer system or on-line at the Java web site: `http://java.sun.com/apis.html`.

10.1 Text Fields

As our first of example of using GUI components we introduce the class `JTextField` from Java's Swing package. An object from the class `JTextField` displays a single line of user-updatable text.

A `JTextField` object differs from objects of objectdraw's `Text` class in two important ways. First, while only the programmer can change the contents of a `Text` object, a user may change the contents of a `JTextField` object by clicking in the field and typing new text. Thus `JTextField`s are very useful for obtaining user input.

Another important difference between the two is the way objects are placed in a window. A `Text` object can be placed at any location desired on a `canvas`. A `JTextField` must be placed in a window using a layout manager.

Figure 10.2 contains the code for a class that displays a window with a text field at the top of the window and a canvas filling the rest of the window. Each time the user clicks on the canvas, the program generates a `Text` object with the contents of the text field. Figure 10.1 displays a window that shows the operation of this program.

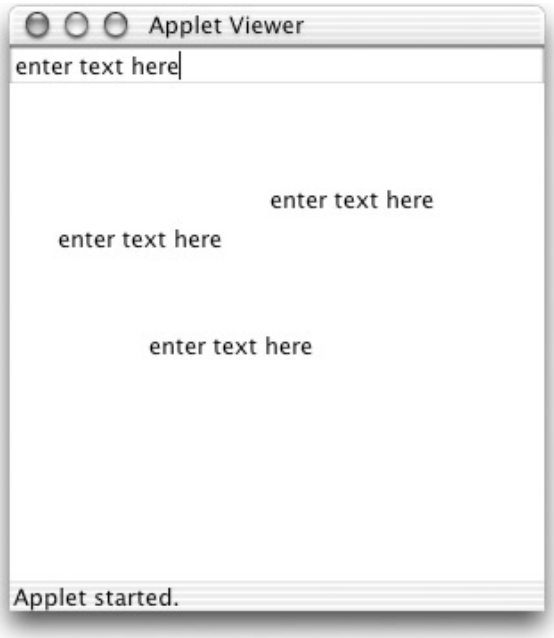

Figure 10.1: JTextField in window.

10.1.1 Creating a Text Field

Text fields are created using the built-in `JTextField` class that is part of Java's Swing package. You can use the `JTextField` class like any other class. For example, the `TextController` class in Figure 10.2 contains the instance variable declaration:

```
private JTextField input;
```

The constructor for `JTextField` that we will use takes a `String` parameter specifying the initial contents appearing in the text field. In our example, the component is created in the `begin` method with the statement:

```
input = new JTextField( "Enter text here" );
```

If you do not want the text field to contain any text initially, you may also use a parameterless constructor, as in:

```
input = new JTextField();
```

To use the `JTextField` class, you must *import* it from the Swing package into any classes that use it. Importing information from a package makes the names in the package accessible to the class to which it is being imported.

```
import objectdraw.*;
import java.awt.*;
import javax.swing.*;

public class TextController extends WindowController {
    private JTextField input;

    public void begin() {
        // create text field
        input = new JTextField( "Enter text here" );

        // Add text field to content pane of window
        Container contentPane = getContentPane();
        contentPane.add( input, BorderLayout.NORTH );
        contentPane.validate();
    }

    // Add new text item where clicked.
    // Get contents from text field
    public void onMouseClick( Location point ){
        new Text( input.getText(), point, canvas );
    }
}
```

Figure 10.2: `TextController` class using `JTextField`

We have two choices in importing classes from a package. We may import only the names of the classes we need or we may import all of the names defined in that package. We import a single name in an `import` declaration by listing the name with the package's name prefixed to it. For example, we import the name of the `JTextField` class from the Swing package by writing:

```
import javax.swing.JTextField;
```

Notice that the full name of the Swing package is `javax.swing`. Alternatively, we can import all of the names in the package by simply writing:

```
import javax.swing.*;
```

The "`*`" at the end indicates that all names from the package `javax.swing` should be imported.

Import statements for a class or interface definition should be placed before the first line of the `class` or `interface` declaration, in the same place we write `import objectdraw.*`. The relative ordering of import statements does not matter in Java.

While we normally omit import statements in our code examples, we have included them in Figure 10.2 and many of the other examples in this chapter in order to make it clear where they belong. In the example we have imported all of the names in packages `objectdraw`, `java.awt`, and `javax.swing`.

10.1.2 Installing a Text Field in a Window

While the construction of a `Text` object makes it appear immediately on the `canvas` used as a parameter, creating a `JTextField` object does not make it appear in the window. To make a component appear in a window, we must add it to the "content pane" of the window.

We can obtain the content pane of a class extending `WindowController` by evaluating the built-in method `getContentPane()`. The statement

```
Container contentPane = getContentPane();
```

in the `begin` method of Figure 10.2 stores the content pane of the window in the variable `contentPane`, which has type `Container`. A `Container` object can hold GUI components like text fields, buttons, and menus. The class `Container` comes from Java's AWT package. To use it we must import `java.awt.Container` or `java.awt.*`, as we did above.

Every `Container` object created by Java comes with a *layout manager*. The layout manager takes care of arranging components in the window according to the instructions of the programmer. Content panes associated with classes extending `WindowController` come with a `BorderLayout` layout manager as the default.

Figure 10.3 shows how components can fit into the different parts of a window using `BorderLayout`. Components may be installed in the center, north, south, east, or west position in a `Container` with this layout manager. Normally, we will avoid adding to the center of the contents pane of a window associated with an extension of `WindowController` because the `canvas` is already installed there.

If you wish to create an applet that does not have the `canvas` already installed in the center position, you may define your class to extend `Controller`, rather than `WindowController`. The class `Controller` was included in the objectdraw library precisely for those situations where the

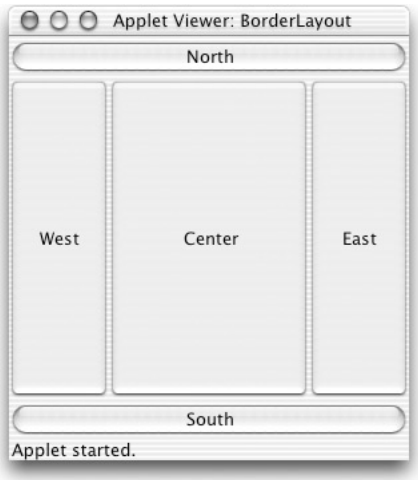

Figure 10.3: Layout of a window using BorderLayout layout manager.

`canvas` was not needed. Of course classes that extend `Controller` no longer suppose the mouse-handling methods like `onMouseClick` because those methods were called in response to mouse actions on the `canvas`.

The method for adding a component to a container that uses `BorderLayout` is

```
someContainer.add( aComponent, position);
```

where `someContainer` has type `Container` and `position` is one of the following five constants:

- `BorderLayout.NORTH`,
- `BorderLayout.EAST`,
- `BorderLayout.SOUTH`,
- `BorderLayout.WEST`, or
- `BorderLayout.CENTER`.

For example, the `input` text field is added to the top edge of the window in the `begin` method when the following statement is executed:

```
contentPane.add( input, BorderLayout.NORTH );
```

When a text field is added in the north or south positions in a `Container` that uses the `BorderLayout` layout manager, the text field will be stretched to fill the entire width of the window. If it is added to the east or west, it will be stretched vertically to reach from the top of the south item to the bottom of the north item. Items added to the center stretch to fill all space between the north, east, south, and west components. We will learn later how to use panels in Java to provide more control over the display of GUI components, but we postpone that discussion for now.

The last thing you should do after adding components to a container is to send the message `validate()` to that container. While most of the time this will have no effect, many times it is necessary to get the system to recompute the size and layout of all of the components in the container. If you leave this off, components might not be displayed properly.

10.1.3 Getting Information from a Text Field

The class `TextController` creates a new `Text` object on the canvas every time the user clicks the mouse on the canvas. The string displayed in the new `Text` object is obtained from `input`, an object of type `JTextField`. The current contents of a text field are obtained by sending it a `getText` message, as is shown in the `onMouseClick` method. It is also possible to send a text field a `setText` message to reset the contents of the field.

In summary, a text field can be a useful way of getting text input from a user into a program. To use a text field you must create it and then add it to the desired location in the window. The `getText` method can be used to retrieve the contents of a text field.

Exercise 10.1.1 *Write a class called* `TextMimic` *that places a text field on the bottom of the window. When the mouse is clicked on the canvas, the program should display the contents of the text field as a* `Text` *object at the location the mouse was clicked. For example, if the text field showed "Hello world." and the mouse was clicked at the location (100, 100), then the program would display a* `Text` *object at the location (100, 100) that read "Hello world."*

10.2 Buttons and Events in Java

In the previous section we showed how to create, install, and get the contents of a `JTextField`. In this section, we show how to use buttons in Java. Buttons require slightly more programming effort than text fields because we want clicks on buttons to trigger program actions. We explain how to accomplish this using Java events and event-handling methods.

As a simple example of the use of a button in a Java program, we will modify the `TextController` class from the previous section so that the canvas is erased whenever a button is pressed. The new `TextButtonController` class is shown in Figure 10.4.

10.2.1 Creating and Installing Buttons

Buttons are represented in the Swing package by the class `JButton`. Buttons may be created and installed in the same way as text fields. Because the button is only referenced in the `begin` method of Figure 10.4, it can be declared as a local variable:

```
JButton clearButton = new JButton( "Clear Canvas" );
```

The constructor takes a `String` parameter representing the label that appears on the button. As with a text field, the button must also be added to the content pane of the window. This time we add it to the south part of the window:

```
contentPane.add( clearButton, BorderLayout.SOUTH );
```

10.2.2 Handling Events

We already know that when the user clicks on the canvas, the code in the `onMouseClick` method is executed. When a user clicks on a button created from the class `JButton`, however, the `onMouseClick` method is not executed. Instead, the operating system begins executing a method, `actionPerformed`, that is designed to handle button clicks.

```
import objectdraw.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TextButtonController extends WindowController
                                  implements ActionListener {
    private JTextField input;

    public void begin() {
        // create text field and button
        input = new JTextField( "Enter text here" );
        JButton clearButton = new JButton( "Clear Canvas" );

        // Add text field and button to content pane of window
        Container contentPane = getContentPane();
        contentPane.add( input, BorderLayout.NORTH );
        contentPane.add( clearButton,BorderLayout.SOUTH );
        contentPane.validate();

        // Set this class to respond to button clicks
        clearButton.addActionListener( this );
    }

    // Add new text item where clicked.
    // Get contents from text field
    public void onMouseClick( Location point ){
        new Text( input.getText(), point, canvas );
    }

    public void actionPerformed( ActionEvent evt ) {
        canvas.clear();
    }
}
```

Figure 10.4: `TextButtonController` class using `JTextField`

To understand how this works, let's back up for a minute to get a better understanding of how Java programs respond to user actions. Java supports a style of programming known as *event-driven programming*. We have been programming in this style with our programs that react to mouse actions. For example, whenever the user clicks the mouse button, it generates an event that results in the `onMouseClick` method being executed by the computer.

A button generates an event of type `ActionEvent` whenever the user clicks on it. This in turn results in the execution of an `actionPerformed` method. For example, the method `actionPerformed` in `TextButtonController` is executed each time the button is pressed. In this example it simply erases the canvas.

Just as the `onMouseClick` method is provided with a parameter representing the location where the mouse was clicked, the `actionPerformed` method is provided with a parameter of type `ActionEvent` that contains information about the object that triggered the event. For example, `evt.getSource()` returns the GUI component that triggered an event, `evt`. While we don't need that information for this example, we will present examples later that do use that information.

There is one extra step that must be taken when handling events in standard Java. In order for a program to respond to an event generated by a GUI component, we must designate one or more objects to be notified when an event is generated by that component.

An object that is to be informed when an event is generated by a component is said to be an event *listener*. Different kinds of events require different types of listeners. The only requirement for an object to be a listener for a particular kind of event is that it promise to provide an appropriate method to be triggered when the event occurs.

Pressing on a button triggers the execution of an `actionPerformed` method. Thus any listener for a button must implement the method:

```
public void actionPerformed( ActionEvent evt );
```

We associate a listener with a button by sending the button an `addActionListener` message, such as:

```
clearButton.addActionListener( actionListenerObject );
```

After this code has been executed, `actionListenerObject` is registered as a listener for `clearButton`, and its `actionPerformed` method is executed each time the button is pressed.

What is the type of the parameter of `addActionListener`? As we have seen, we only require that it provide an `actionPerformed` method. The Java AWT package contains an interface `ActionListener` whose only method declaration is the `actionPerformed` method. Thus any object that implements the `ActionListener` interface can be used as a parameter for `addActionListener`.

While we could design a class that creates objects that are only used as listeners, for simplicity we will generally handle interaction with GUI components similarly to the way in which we have been handling mouse actions. That is, we will have the window class itself contain the method that responds when the user interacts with a GUI object. Thus we write

```
clearButton.addActionListener( this );
```

in the `begin` method of `TextButtonController`. That indicates that the `TextButtonController` object will do the listening. In order for this to be legal, the class `TextButtonController` must implement the interface `ActionListener` and include a method `actionPerformed`. The `actionPerformed` method in Figure 10.4 erases the canvas.

Both `ActionEvent` and `ActionListener` are in the `java.awt.event` package; any program using them must include the following import statement:

```
import java.awt.event.*;
```

Warning! *Do not make the mistake of trying to handle a button click by writing an* `onMouseClick` *method. That method is only executed when the user clicks on the canvas, not on a GUI component. The* `onMouseClick` *method handles clicks on the canvas, while* `actionPerformed` *handles clicks on a button.*

Also don't forget to add a listener to buttons in your program. If you leave that line out of your class the system won't complain, but will just ignore all clicks on the button.

10.3 Checklist for Using GUI Components in a Program

We have now seen examples using both text fields and buttons. With that background, we can now summarize the actions necessary to create and use a GUI component in a class extending `WindowController`, where that class also serves as the listener for the component.

1. **Create the GUI component**:

   ```
   input = new JTextField( "enter text here" );
   clearButton = new JButton( "Clear Canvas" );
   ```

2. **Add the component to the content pane of the `WindowController` extension and validate it**:

   ```
   Container contentPane = getContentPane();
   contentPane.add( input, BorderLayout.NORTH );
   contentPane.add( clearButton,BorderLayout.SOUTH );
   contentPane.validate();
   ```

3. **If a `WindowController` extension is to respond to events generated by the component,**

   (a) **Add `this` as a listener for the component**:

   ```
   clearButton.addActionListener ( this );
   ```

   (b) **Add a declaration that the `WindowController` extension implements the appropriate listener interface**:

   ```
   public class TextButtonController extends WindowController
                                implements ActionListener {
   ```

   (c) **Add to the `WindowController` extension the event-handling method promised by the listener interface**:

```
public void actionPerformed( ActionEvent evt ) {
    ...
}
```

While different kinds of GUI components generate different kinds of events requiring different event-handling methods and listener types, the checklist above summarizes the steps a programmer must take in order to install and use any kind of GUI component. Always make sure that you have taken care of each of these requirements when using GUI components.

Exercise 10.3.1 *Write a class called* `ClickMe` *that places a button on the bottom of the window. The button should be labeled "Click Me." When the button is clicked, a* `Text` *object should display the number of times the user has clicked. For example, after the user has clicked twice, the canvas should read, "You have clicked 2 times." Be sure to follow steps* `1` *through* `3` *to add the button.*

Exercise 10.3.2 *Write a class called* `RandomCircles` *that places a button on the top of the window. The button should be labeled "Draw a circle." When the button is clicked, the program should draw a circle of a random size (between 10 and 100 pixels in diameter) at a random location on the canvas. Be sure to follow steps* `1` *through* `3` *to add the button.*

10.4 Combo Boxes

A popular GUI component used in Java programs is a pop-up menu, called a *combo box* in Java's Swing package. In Figure 10.5 we introduce a slight variant of the `FallingBall` class from Figure 8.3 in Chapter 8. In this new version we have added a new method `setSpeed` that allows us to change how far the ball falls between pauses. Using a large value for the integer parameter makes the ball appear to move faster, while a small value makes it appear to move slower.

We will write a class extending `WindowController` to handle user interactions with the ball. As with the previous case, when the user clicks on the canvas, the program will create a ball that will begin falling. However, we will also introduce a combo box (menu) to allow the user to change the speed at which the ball will fall. This will be accomplished in the class `MenuBallController` in Figure 10.7. The combo box will include three alternatives, "Slow", "Medium", and "Fast." A picture showing the window with the combo box at the bottom is given in Figure 10.6.

Let's follow through the checklist from the previous section to make sure that we do everything necessary to install properly and use a combo box GUI component generated from the class `JComboBox`. As we do this, we will also highlight those places where combo boxes are handled differently from buttons or text fields.

1. **Create the combo box**: Combo boxes are a bit more complicated than buttons in that we must add the selection options to the combo box once it has been constructed. The relevant code in the class is:

```
speedChoice = new JComboBox();

speedChoice.addItem ( "Slow" );
speedChoice.addItem ( "Medium" );
speedChoice.addItem ( "Fast" );
```

```
public class FallingBall extends ActiveObject {
    // constants omitted
    private FilledOval ball;     // Image of ball as circle
    private int speed;           // current speed of ball

    // Draw ball at location and w/speed given in parameters
    public FallingBall( Location ballLocation, int initSpeed,
                        DrawingCanvas canvas ) {
        ball = new FilledOval( ballLocation, BALLSIZE, BALLSIZE, canvas );
        speed = initSpeed
        start();
    }

    // Move ball down until off of canvas
    public void run() {
        while ( ball.getY() < canvas.getHeight() ) {
            ball.move( 0, speed );
            pause( DELAY_TIME );
        }
        ball.removeFromCanvas();
    }

    // reset speed of ball
    public void setSpeed( int newSpeed ) {
        speed = newSpeed;
    }
}
```

Figure 10.5: `FallingBall` class

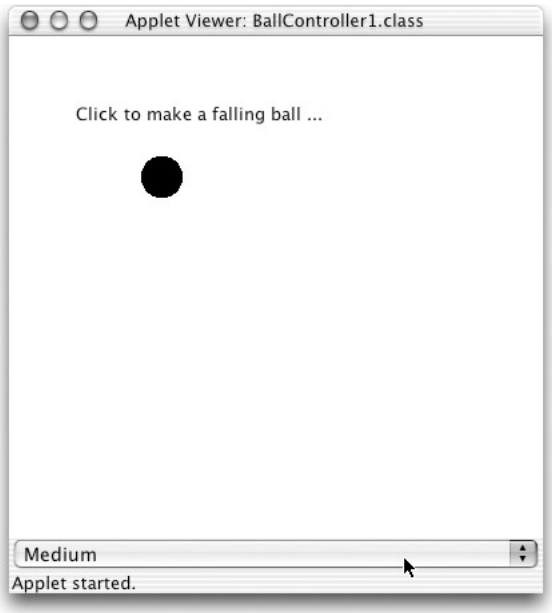

Figure 10.6: Menu created by `MenuBallController`.

```
speedChoice.setSelectedItem( "Medium" );
```

The first line creates the combo box and associates it with variable `speedChoice`. The next three lines use the `addItem` method to add the items "Slow", "Medium", and "Fast" to the combo box.

Items appear on the pop-up menu of the combo box in the order that they are added, with the first item added showing when the combo box is first displayed. Thus "Slow" would normally be displayed when the combo box first appears. If we want to have a different item showing initially, we can send either a `setSelectedIndex` or a `setSelectedItem` message to the combo box.

With `setSelectedIndex`, we must provide an `int` parameter that specifies the index (i.e., the number) of the item that is to be displayed. Indices in Java always start with 0, so in our speedChoice example, adding the statement

```
speedChoice.setSelectedIndex( 1 );
```

would result in the item "Medium" being displayed, as in Figure 10.6.

If, as in the actual code, the `setSelectedItem` method is used, then the parameter should be the actual item to be selected. Thus executing the statement

```
speedChoice.setSelectedItem( "Medium" );
```

```
import objectdraw.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MenuBallController extends WindowController
                              implements ActionListener {
    // constants omitted
    private FallingBall droppedBall; // the falling ball

    private JComboBox speedChoice;   // Combo box to select ball speed
    private int speed;               // Current speed setting

    // display instructions and combo box
    public void begin() {
        new Text( "Click to make a falling ball...", INSTR_LOCATION, canvas );

        speed = SLOW_SPEED;

        speedChoice = new JComboBox();         //Create combo box

        speedChoice.addItem ( "Slow" );        // Add 3 entries
        speedChoice.addItem ( "Medium" );
        speedChoice.addItem ( "Fast" );
        speedChoice.setSelectedItem( "Medium" ); // Display "Medium" initially

        speedChoice.addActionListener ( this );    // this class is listener

        Container contentPane = getContentPane(); // Add combo box to south
        contentPane.add( speedChoice, BorderLayout.SOUTH );
        contentPane.validate();
    }

    // make a new ball when the player clicks
    public void onMouseClick( Location point ) {
        droppedBall = new FallingBall( point, speed, canvas );
    }

    // reset ball speed from combo box setting
    public void actionPerformed( ActionEvent evt ) {
        Object newLevel = speedChoice.getSelectedItem ();
        if ( newLevel.equals( "Slow" )) {
            speed = SLOW_SPEED;
        } else if ( newLevel.equals( "Medium" )) {
            speed = MEDIUM_SPEED;
        } else if ( newLevel.equals( "Fast" )) {
            speed = FAST_SPEED;
        }
        if ( droppedBall != null ) {
            droppedBall.setSpeed( speed );
        }
    }
}
```


Figure 10.7: `MenuBallController` class

also results in the item "Medium" being displayed.

2. **Add the combo box to the content pane of the `WindowController` extension and validate it**:

```
Container contentPane = getContentPane();
contentPane.add( speedChoice, BorderLayout.SOUTH );
contentPane.validate();
```

3. **Because we need a listener**:

   (a) **Add `this` as a listener for the combo box**: Like buttons, combo boxes require a class that implements the `ActionListener` interface to handle the events generated. The method `addActionListener` is used to add a listener

```
speedChoice.addActionListener( this );
```

   (b) **Add a declaration that the `WindowController` extension implements the appropriate listener interface**: `ActionListener` is the appropriate interface for a combo box:

```
public class MenuBallController extends WindowController
                                 implements ActionListener {
```

   (c) **Add to the `WindowController` extension the event-handling method promised by the listener interface**: The `ActionListener` interface includes the method `actionPerformed`, which takes a parameter of type `ActionEvent`:

```
public void actionPerformed( ActionEvent evt ) {
    ...
}
```

There is little difference between installing and using a combo box and a button. The body of the `actionPerformed` method resets the speed of the last ball dropped as well as balls to be dropped in the future. To make these changes to the speed, it needs a way to determine the current setting of a combo box. The class `JComboBox` provides two methods that return the combo box's current setting. The first is `getSelectedItem()`, which returns the object that the user chose from the combo box. The other is `getSelectedIndex()`, which returns the index of the item chosen (starting from 0 as usual).

In this case, it is reasonable to use `getSelectedItem()` to return the item selected by the user from the combo box, storing the value in the variable, `newLevel`. The method `getSelectedItem` has return type `Object`. Thus the local variable `newLevel` in `itemStateChanged` must have type `Object`.

`Object` is a built-in Java class that represents all objects. It includes all values aside from those belonging to primitive types like `int`, `double`, and `boolean`, and supports the methods:

```
public boolean equals(Object other);

public String toString();
```

If not redefined in a class, the method `equals` returns `true` if the receiver and the parameter are the *same* object. That is, it behaves like "==". However, it is a good idea for classes to provide different behavior where appropriate. For example, the `String` class has redefined `equals` so that it returns true if `other` is a `String` with the same sequence of characters as the receiver.

The method `toString` returns a `String` representing the object. If not redefined in a class, it simply returns the name of the type along with an integer. For example, an object from class `Letters` might return `Letters@858610` when sent the `toString` message. It is again a good idea to redefine this method in classes so that you get back a reasonable `String` representation of an object. For example, if you send a `toString` message to `Color.red`, it will return "java.awt.Color[r=255,g=0,b=0]". Sending `toString` to a `FramedRect` will return a string that provides information about its location, dimensions, and color. Not surprisingly, sending a `toString` method to an object of type `String` returns a copy of the receiver, as it is already a string.

`Object` is used as the return type of `getSelectedItem` because any type of object may be inserted as an item in a combo box. That is, the parameter in method `addItem` is of type `Object`. The actual text displayed for that item in the combo box is the string obtained by sending a `toString` message to that object.

Even though we don't know what kind of object will be returned from `getSelectedItem`, we can still use the `equals` method to compare the value returned from the invocation of `getSelectedItem` with the items inserted into the combo box. In our example these are all strings and the comparisons work as expected.

Exercise 10.4.1 *Under what circumstances might* ***`droppedBall`*** *have the value* ***`null`*** *in the execution of the method* ***`itemStateChanged`****?*

Exercise 10.4.2 *If you click the mouse in the window several times in rapid succession, multiple balls will be created and will fall toward the bottom of the window. If you use the combo box to change the speed, only the last ball created will change speed. Explain why.*

Exercise 10.4.3 *Add a new speed setting, "Very Slow", as the first entry in the combo box. However the initial setting should have the ball moving at the "Slow" speed and have the "Slow" item showing initially on the combo box.*

Exercise 10.4.4 *Write a class called* `BoxColor` *that initially draws a black* `FilledRect` *on the canvas. Add a combo box to the bottom of the window. The combo box should include the items "red," "yellow," "green," "blue," and "black." When a color is chosen from the combo box, the color of the* `FilledRect` *should change appropriately.*

10.5 Panels and Layout Managers

We now know how to add a text field, a button, or a combo box to a window. However, the buttons and combo boxes may not look as expected because they are distorted to fill the entire width of the window.

Another problem is that we often want to insert two or more components in the same part of the window. For example it may be easier for the user if all of the components provided to control a program are adjacent. However, if we insert two GUI components, say a combo box and a button, to the same part of the window (e.g., the south side), the second one will simply, and unfortunately, appear in place of the first.

In this section we will learn how to solve these problems and make more attractive user interfaces by using panels and by introducing several different layout managers.

10.5.1 Panels

We begin by solving the following simple problem. Rather than using a combo box to set the speed of the falling ball, as in the `MenuBallController` class, we would like to have three buttons to set the speed to three values.

We could add each button to a different part of the window. For example, we could put one each on the east, north, and west sides of the window. However, this would both look strange and be inconvenient for the user. We would rather place all three buttons on the south side of the window, as shown in Figure 10.8, but, as we just remarked, only one component can be placed in each of the four available positions in the window.

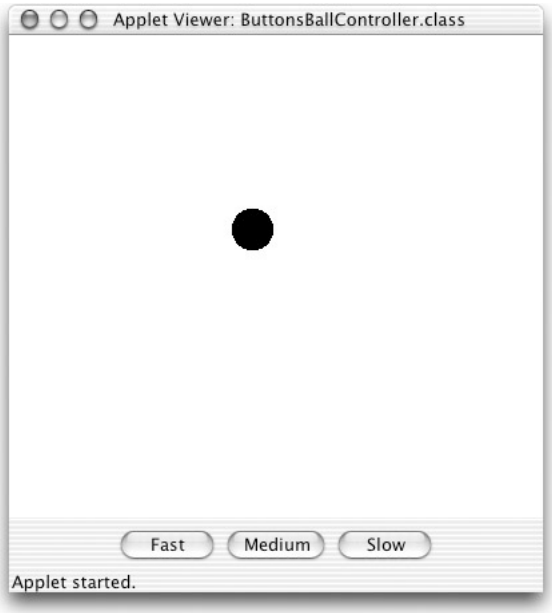

Figure 10.8: Using a panel to group GUI components.

To help solve problems like this, Swing provides a component called a `JPanel`. A panel can contain GUI components just like a window does, but can also serve as a component in the same

way as a button or combo box. Panels are typically used to organize the window into collections of subcomponents. In particular, if we wish to put more than one GUI item in the same part of a window, we can create a panel, add the GUI components to the panel, and then add the panel to the window.

We illustrate this in Figure 10.9. In the `begin` method of the `ButtonsBallController` class we create a panel with three buttons that will be inserted at the bottom (i.e., south side) of the window. Figure 10.8 shows the resulting window with three buttons at the bottom.

A panel uses a different layout manager, `FlowLayout`, than classes that extend `WindowController`. The `FlowLayout` manager lays out its components from left to right across the panel. If there is insufficient space to fit all of the components in a single row, new rows are added as necessary. Notice that each of the buttons in Figure 10.8 is just large enough to display its label, rather than being stretched to fill up all of the available space.

In Figure 10.9, the second line of the `begin` method declares and initializes a local variable, `southPanel`, to be a `JPanel`. It is created with a parameterless constructor. The statements,

```
southPanel.add( fastButton );
southPanel.add( mediumButton );
southPanel.add( slowButton );
```

add the three buttons to `southPanel`. Because the buttons are displayed from left to right in the order they are added, there is no parameter specifying where the button goes in the panel.

Finally, the code:

```
Container contentPane = getContentPane();
contentPane.add( southPanel, BorderLayout.SOUTH );
contentPane.validate();
```

adds `southPanel` to the content pane of the window and then validates the content pane. Because the window uses `BorderLayout`, the method call includes the constant `BorderLayout.SOUTH`, specifying in which part of the window the panel is inserted.

There is still one problem we must solve. When the user clicks on one of the buttons, the `actionPerformed` method of the class is executed because `this` was declared to be a listener for each of the three buttons. Inside the method we need to know which button was pressed so that we can set the speed appropriately. The `actionPerformed` method is given in Figure 10.10.

We can find the source of an event by sending a `getSource` message to the event parameter. We can then compare the source of the event with each of the three buttons to see which was responsible for the event. The `if-else` statement in Figure 10.10 compares the source of the event with the buttons to determine which button was pressed. Once this has been determined, the ball's speed can be updated.

Each of Java's event types supports a `getSource` method. Thus if you have multiple components in a program that call the same method to handle events, you can send a `getSource` message to the event parameter to determine which component generated the event.

As with the earlier `getSelectedItem` method, the `getSource` method returns a value of type `Object`. Thus if we save the results of evaluating `evt.getSource()` in a temporary variable, that variable must have type `Object` in order to pass the type-checker. As before, there is no difficulty in comparing the returned value with other objects, in this case the buttons in the window.

In the same way that we added a panel to the south side of the window, we could add panels to the north, east, and west sides. With more complex collections of GUI components, one can

```
import objectdraw.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonsBallController extends WindowController
                                   implements ActionListener {
    // constants omitted

    private FallingBall droppedBall; // the falling ball

    private JButton fastButton,       // Buttons to control speed
                    mediumButton,
                    slowButton;

    private int speed;                // Current speed setting

    // display buttons
    public void begin() {
        speed = SLOW_SPEED;

        JPanel southPanel = new JPanel();

        fastButton = new JButton( "Fast" );
        mediumButton = new JButton( "Medium" );
        slowButton = new JButton( "Slow" );

        fastButton.addActionListener( this );
        mediumButton.addActionListener( this );
        slowButton.addActionListener( this );

        southPanel.add( fastButton );
        southPanel.add( mediumButton );
        southPanel.add( slowButton );

        Container contentPane = getContentPane();
        contentPane.add( southPanel, BorderLayout.SOUTH );
        contentPane.validate();
    }

    // make a new ball when the player clicks
    public void onMouseClick( Location point ) {
        droppedBall = new FallingBall( point, speed, canvas );
    }
```

Figure 10.9: `ButtonsBallController` with multiple buttons in a panel. Part 1.

```
    // set new speed when the player clicks a button
    public void actionPerformed( ActionEvent evt ) {
        if ( evt.getSource() == slowButton ) {
            speed = SLOW_SPEED;
        } else if ( evt.getSource() == mediumButton ) {
            speed = MEDIUM_SPEED;
        } else {
            speed = FAST_SPEED;
        }

        if ( droppedBall != null ) {
            droppedBall.setSpeed( speed );
        }

    }

}
```

Figure 10.10: `ButtonsBallController` with multiple buttons in a panel. Part 2.

even add panels which themselves contain other panels. Because our focus here is on programming rather than the layout of GUI components, we will not explore these possibilities further. However, we will discuss layout managers briefly in the next section.

Exercise 10.5.1 *Suppose we only added one button to* `southPanel`*. How would the appearance of the window differ from that that obtained if we just insert the button directly in the south of the window?*

Exercise 10.5.2 *Variable* `southPanel` *is declared to be a local variable of the* `begin` *method rather than an instance variable. Why?*

Exercise 10.5.3 *Write a class called* `ColorBox` *that initially draws a black* `FilledRect` *on the canvas. Add four buttons to the bottom of the window using a panel. The buttons should be labeled "Black," "Red," "Yellow," and "Blue." When a button is clicked, the color of the* `FilledRect` *should change appropriately.*

Exercise 10.5.4 *Revise* `ButtonsBallController` *to add a combo box to change the color of the ball. Put the combo box in a new panel in order to have the combo box look better (try it both with and without the panel to see the difference). Place the panel on the north of the window.*

10.5.2 More on Layout Managers

You may be wondering why we don't just specify the size and location of components in the window, as we did with graphics objects on the canvas, rather than depending on layout managers and panels to arrange them. One reason is that the user can resize the window at any time. It would be very inconvenient to have the user resize the window so that some of the components were no longer

visible. Of course, if the user makes the window too small, it is possible that there really is not room for all of the components. However, the window managers do their best to display all of the components if that is at all possible. Play around with some of the example programs using GUI components to see that this is indeed the behavior.

Layout managers determine how GUI components are displayed in a window or panel. Container classes like panels and the content panes of windows have default layout managers, but the programmer can change them before adding components by sending the container a `setLayout` message.

The `BorderLayout` manager, the default layout manager for the content pane of classes extending `WindowController`, enables a program to add new components (including panels) to the north, south, east, or west portions of a container with that layout manager. As stated earlier, the `canvas` has already been added to the center of extensions of `WindowController`, so it is important not to add a component in the center. If you wish to insert a component in the center, you should extend `Controller`, instead.

The `FlowLayout` manager, which is the default for panels, enables a program to add an unlimited number of components to a container with that layout manager. These components are laid out from left to right in the order in which they are added, in the same way that you add new text with a word processor. If there is not enough room for all of the components, new rows are added, which are also filled from left to right.

The layout manager of a container, `someContainer`, can be changed by sending it a `setLayout` message. For example, the layout manager for a content pane, `contentPane`, of a window can be changed to be FlowLayout by invoking:

```
contentPane.setLayout( new FlowLayout() );
```

Similarly, the layout manager of `somePanel` can be set to be `BorderLayout` by invoking:

```
somePanel.setLayout( new BorderLayout() );
```

The `GridLayout` manager can be used to divide a container into equally sized parts. A container with this layout manager is divided into a group of equally sized cells arranged in a grid. Like the `FlowLayout` manager, components are displayed from left to right across each row in the order in which they are added. The `GridLayout` manager differs from the `FlowLayout` manager by forcing each component to take the same amount of space and by being limited to a predetermined number of rows and columns.

Suppose `southPanel` is a panel that we wish to divide into a grid with `numRows` rows and `numCols` columns. This can be accomplished by executing:

```
southPanel.setLayout( new GridLayout( numRows, numCols ) );
```

Suppose we modify the `ButtonsBallController` class in Figure 10.9 by adding the statement:

```
southPanel.setLayout( new GridLayout( 1, 3 ) );
```

after the panel has been created, but before any buttons have been added. As before, the three buttons will be added in order, but this time they are stretched so that each takes up one-third of the space in the panel. See the picture in Figure 10.11.

There are other layout managers available for Java, but we do not cover them here. Instead we turn our attention to other GUI components.

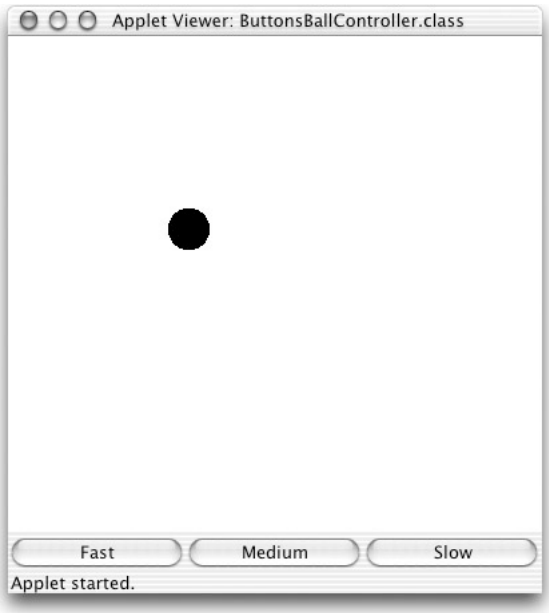

Figure 10.11: A panel with `GridLayout` manager to group buttons.

Exercise 10.5.5 *How would you change the statement*

```
southPanel.setLayout (new GridLayout(1,3));
```

used to make the three-button display in Figure 10.11 to make it display three buttons on top of each other instead of side by side?

10.6 Other GUI Components

In this section we look at some of the other GUI components that you may find useful. We do not attempt to list all of them, but instead include a representative sample.

10.6.1 Sliders

Sliders can be used to select values in a range. A slider, constructed from class `JSlider`, consists of a horizontal or vertical line with a marker that can be dragged to indicate a value between the maximum and minimum values represented by the slider. Figure 10.12 shows the window for a program that uses a slider to determine the speed of a falling ball.

Using buttons or a combo box to select speeds can be inconvenient because they only allow us to choose from a very limited selection of speeds. While we could certainly add more choices to the combo box, it makes more sense to use a slider to allow the user to smoothly choose from a range of values. The code for the class `SliderBallController` can be found in Figure 10.13.

For the most part, the steps in constructing and using a slider are very similar to those required to construct and use combo boxes and buttons. The only real differences are in the actual constructor for the slider, which has a number of parameters to control its appearance and operation,

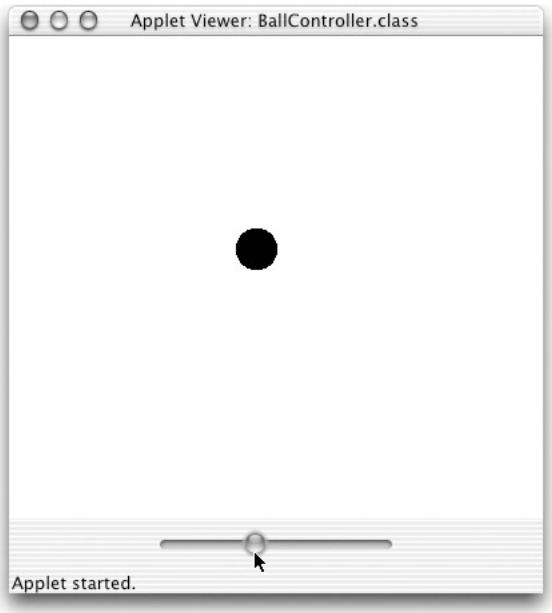

Figure 10.12: A window with a `Slider` to set the speed.

the listener name, and the method used to get the value represented by the slider. The steps are as follows:

1. **Create the slider**: This is taken care of in the first line of the `begin` method. The constructor takes four parameters.

   - The first parameter is a constant that determines whether the slider is oriented horizontally or vertically. The possible values are `JSlider.HORIZONTAL` and `JSlider.VERTICAL`.
   - The second and third parameters are used to indicate the minimum and maximum integer values that can be represented by the slider.
   - The fourth parameter sets the initial value of the slider.

   The slider in `SliderBallController` is initialized to be horizontal, representing the range of values between `SLOW_SPEED` and `FAST_SPEED`, and with initial value set to `SLOW_SPEED` by the following call of the constructor.

   ```
   speedSlider = JSlider( JSlider.HORIZONTAL, SLOW_SPEED, FAST_SPEED, SLOW_SPEED
   );
   ```

2. Add the slider to the content pane of the `WindowController` extension and validate it.

   ```
   Container contentPane = getContentPane();
   contentPane.add( speedSlider, BorderLayout.SOUTH );
   contentPane.validate();
   ```

```
import objectdraw.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class SliderBallController extends WindowController
                                  implements ChangeListener {
    // constants omitted
    private FallingBall droppedBall; // the falling ball

    private JSlider speedSlider;     // slider to set speed

    private int speed = SLOW_SPEED;  // speed of ball

    // create and install GUI items
    public void begin() {
        speedSlider = JSlider( JSlider.HORIZONTAL, SLOW_SPEED,
                               FAST_SPEED, SLOW_SPEED );
        speedSlider.addChangeListener ( this );

        Container contentPane = getContentPane();
        contentPane.add( speedSlider, BorderLayout.SOUTH );
        contentPane.validate();
    }

    // make a new ball when the player clicks
    public void onMouseClick( Location point ) {
        droppedBall = new FallingBall( point, speed, canvas );
    }

    // Get new speed from slider
    public void stateChanged( ChangeEvent evt ) {
        speed = speedSlider.getValue();
        if ( droppedBall != null ) {
            droppedBall.setSpeed( speed );
        }
    }
}
```

Figure 10.13: `SliderBallController` class with slider

3. **Because a listener is required**:

   (a) **Add `this` as a listener for the slider**: Sliders require change listeners, so we write:

   ```
   speedSlider.addChangeListener( this );
   ```

   (b) **Add a declaration that the `WindowController` extension implements the appropriate listener interface**.

   ```
   public class SliderBallController extends WindowController
                                      implements ChangeListener {
   ```

   (c) **Add to the `WindowController` extension the event-handling method promised by the listener interface**. The event-handling method for sliders is `stateChanged`. The method can obtain the value represented by the slider by sending a `getValue` message. In the example, the `stateChanged` method uses that value to reset the speed of the ball most recently dropped and to remember that speed for balls created subsequently.

   ```
   public void stateChanged( ChangeEvent evt ) {
       speed = speedSlider.getValue ();
       if ( droppedBall != null ) {
           droppedBall.setSpeed( speed );
       }
   }
   ```

Because the `ChangeEvent` class is new to Swing, it must be imported from the `javax.swing.event` package.

Exercise 10.6.1 *Write a class called* `SquareSizer` *that uses a slider to determine the size of a square* `FilledRect`*. The* `FilledRect` *should initially be 100 pixels by 100 pixels. The slider should allow the user to change the size of the box to any size between 10 and 300 pixels.*

Exercise 10.6.2 *Write a class called* `RectangleSizer` *that uses two sliders to determine the size of a* `FilledRect`*. One slider should control the length of the box and the other should control the height of the box. Both sliders should have an upper bound of 300 pixels and a lower bound of 10 pixels. The box should start as a 100 by 100 pixel square.*

10.6.2 Labels

One problem with the program above is that it is not at all clear to the user what the slider is for. We should label it to make it clear to the user what it controls. We might be tempted to use an object of the class `Text` to handle this, but `Text` items may only be put on the canvas. Instead we would like to have the label next to the slider.

We can add a label to the left of the slider in our example by adding a panel to the bottom of the window with both the label and the slider. We want the label to show the current speed once the slider has been adjusted. Figure 10.14 shows the window with this label.

The Java Swing package includes a `JLabel` class that does exactly what we want. It is a GUI component that contains a single line of read-only text. It is a passive component and is not

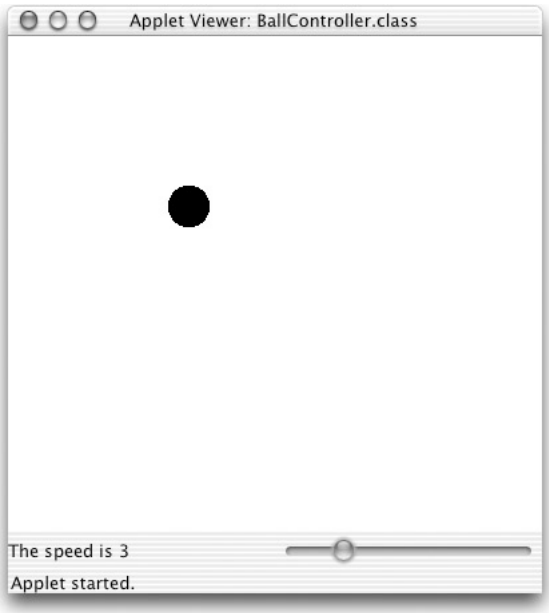

Figure 10.14: JSlider with JLabel in JPanel.

designed to respond to user interactions, so no event listener is required. Instead we simply create the label and install it in the window or panel where we want it to be.

The code for the class `LabelBallController` is shown in Figure 10.15. The `begin` method creates the label, while the `stateChanged` method changes the label as well as the speed.

There are several constructors for labels. The simplest constructor just takes the string to be displayed as a parameter.

```
new JLabel( labelString )
```

An alternative version allows the user to specify whether the label is left or right justified or centered:

```
new JLabel( labelString, justification )
```

where `justification` is one of the following constants: `JLabel.LEFT`, `JLabel.RIGHT`, or `JLabel.CENTER`.

We can change the text on the label by sending it a `setText` message. In the class `LabelBallController`, `setText` is used to update the text in `SpeedLabel` with the new value of the speed every time the user changes the slider. Though we do not need it for this example, the `JLabel` class also supports a `getText` method.

Exercise 10.6.3 *Revise your* `RectangleSizer` *class so that each slider has a corresponding label. The labels should display the height and width of the box and should change as the dimensions of the box change.*

```
import objectdraw.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class LabelBallController extends WindowController
                                 implements ChangeListener {
    ...
    private JLabel speedLabel;

    // create and install GUI items
    public void begin() {
        JPanel southPanel = new JPanel();

        // Create label and slider, and add to southPanel
        speedLabel = new JLabel( "Use the slider to adjust the speed" );
        speedSlider = new JSlider( JSlider.HORIZONTAL, SLOW_SPEED,
                                   FAST_SPEED, SLOW_SPEED );
        southPanel.add( speedLabel );
        southPanel.add( speedSlider );

        // add the panel to south of window
        Container contentPane = getContentPane();
        contentPane.add( southPanel, BorderLayout.SOUTH );
        contentPane.validate();

        // Add listener only for slider
        speedSlider.addChangeListener ( this );
    }

    //Make a new ball when the user clicks
    public void onMouseClick( Location point ) {
        droppedBall = new FallingBall( point, speed, canvas );
    }

    // Update speed from slider
    public void stateChanged( ChangeEvent evt ) {
        speed = speedSlider.getValue ();
        if ( droppedBall != null ) {
            droppedBall.setSpeed( speed );
        }
        speedLabel.setText( "The speed is " + speed );
    }
}
```

Figure 10.15: `LabelBallController` class with label showing speed

10.6.3 JTextField and JTextArea

The first example of this chapter used text fields to hold user input. Text input can be supported with both the `JTextField` and `JTextArea` classes. An object from the class `JTextField` displays a single line of user-updatable text, while a `JTextArea` can display multiple lines of user-updatable text. Both `JTextField` and `JTextArea` components differ from `JLabel` components by allowing the user to directly edit the information displayed.

JTextField

The most common constructor for `JTextField` is the following:

```
new JTextField( initialContents, numberOfColumns );
```

where `initialContents` is a string representing the text shown when the `JTextField` is first displayed and `numberOfColumns` is an `int` representing the desired width of the text field in terms of the number of characters of text to be displayed. You should be aware that the specification of `numberOfColumns` is an approximation, as the width of characters differs unless you are using a monospaced font. Moreover, if a text field is inserted in a container using the `BorderLayout` manager (as it was in the first example of this chapter), the `numberOfColumns` parameter is generally ignored as the text field is stretched or shrunk to fit the space provided.

It is also possible to use one of the following two constructors if you do not wish to specify the number of columns or the initial contents of the text field:

```
new JTextField( initialContents );
new JTextField( numberOfColumns );
```

There are two main ways of associating events with changes to text fields. The simplest is to display a button next to the text field and have the user click on the button when finished editing the field. Alternatively, interacting with the text field can generate events.

The user triggers an event from a text field by hitting the enter or return key. For example, the user might type in a response to a question, hitting the return key at the end of the response. This generates an `ActionEvent` just like a button. To use a text field in this way, the user must add an `ActionListener` to the field, have the listener implement the `ActionListener` interface, and provide an `actionPerformed` method in the listener.

As noted earlier in the chapter, the actual text displayed in an object of class `JTextField` can be obtained by sending it a `getText` message, which will return the `String` displayed in the field. Similarly, the program can display new text in the text field by sending the object a `setText` message with the string to be displayed as a parameter.

JTextArea

The `JTextArea` class is similar to `JTextField`, but objects of this class can represent multiple lines of text. A window with a `JTextArea` is displayed in Figure 10.16.

The most common constructor for this class is:

```
new JTextArea( initialContents, numRows, numCols );
```

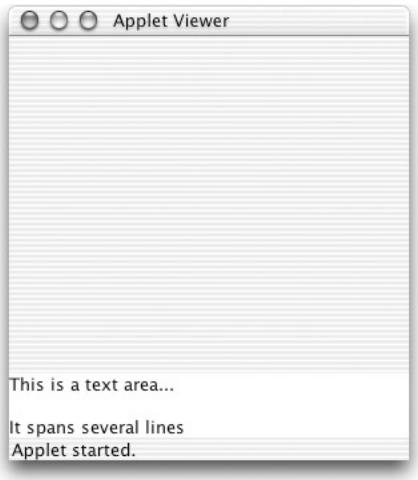

Figure 10.16: JTextArea in window.

where `initialContents` is a `String` representing the text shown when the text area is first displayed, and `numRows` and `numCols` are `int`s representing the desired number of rows and columns of text shown in the text area. Because a `JTextArea` contains multiple lines of data, hitting the return key will not generate an event. Instead, events triggered by other components may result in executing code that includes message sends to the text area, thus obtaining the current contents. Objects from the class `JTextArea` support methods `getText` and `setText` that behave the same way as those for `JTextField`. An `append` message can also be used to add a string to the end of whatever is currently displayed in the text area.

Often we cannot predict in advance how much information will be contained in a `JTextArea`. If more information is contained than will fit in the given space then some information may not be displayed. So that all of the information can be seen, scrollbars may be associated with a text area by wrapping the text area with a `JScrollPane`. The following example illustrates this, where we have used a constructor for `JTextArea` that only provides the desired number of rows and columns:

```
simpleTextArea = new JTextArea( 30, 20 );
JScrollPane scrollableTextArea = new JScrollPane( simpleTextArea );
somePanel.add( scrollableTextArea );
```

where `simpleTextArea` is an instance variable and `somePanel` has been declared elsewhere as a `JPanel`. As before, messages are sent to the text area, not the scroll pane.

10.7 Handling Keystroke and Mouse Events

Keystrokes and mouse actions generate events that can be handled by the appropriate kind of listener. In this section we discuss how to handle these events and discuss how the objectdraw package has so far provided support for handling mouse actions.

10.7.1 Keystroke Events

We can write Java programs so that every keystroke made by the user generates an event that can be handled by the program. This can be useful in processing input one character at a time or in using keyboard input to control actions in games. For example, in a game that controls a rocket ship one could use the arrow keys on the keyboard to change the direction of movement of the ship or use the space bar to trigger an action such as shooting a missile or putting up defense shields.

Keyboard events are generally associated with whichever window is topmost on the screen.[1] Because the window already exists, we can skip the usual steps of creating and installing a component. Instead all we need to do is associate a listener with the keyboard events, and make sure the listener has the appropriate methods to handle key events.

If, as usual, we let the `WindowController` extension be the listener, then we need only include the following statement in the `begin` method:

```
this.addKeyListener( this );
```

While we have normally omitted occurrences of `this` as the receiver of a message, we include it here to emphasize that the key events are coming via the active window.

Because of problems and inconsistencies between some applet viewers and web browsers, you may also have to add a key listener to the canvas:

```
canvas.addKeyListener( this );
```

Finally, we need to inform the system that we wish our applet to gain the "focus". When a component has the "focus", the key events are directed to it. We can request the focus by executing:

```
requestFocusInWindow();
```

Including all three of these statements in a `begin` method should ensure that key events are picked up properly and passed to the listener once the user clicks on the canvas to make sure the applet's window has the focus.[2]

The next step is to declare that the class implements the `KeyListener` interface:

```
class KeyBallController ... implements KeyListener { ... }
```

The `KeyListener` interface contains three methods:

```
public void keyTyped( KeyEvent evt )
public void keyPressed( KeyEvent evt )
public void keyReleased( KeyEvent evt )
```

[1] Keyboard events can also be associated with other components, such as `JTextField`s and `JTextArea`s, but we won't deal with that here.

[2] Unfortunately, on some Windows systems, the focus is not correctly transfered when the user clicks on the applet's window. If the above does not work on your system, also add the following command, which we do not explain further:

```
canvas.addMouseListener (new MouseListener () {
    public void mouseClicked (MouseEvent event) {
        ((JDrawingCanvas)canvas).requestFocusInWindow();
    }
}
```

These three methods behave similarly to `onMouseClick`, `onMousePress`, and `onMouseRelease`. The first is invoked when a key is pressed and then released. The second is invoked when a key is pressed, and the third is invoked when the key is released.

For simple applications only one of the three may be needed. However, the listener object must support all three methods. If only one is really needed, then the other two may be included with empty bodies. Thus if they are invoked, the method finishes immediately without performing any actions.

The two most useful methods available on objects of type `KeyEvent` are `getKeyChar()` and `getKeyCode()`. The first of these returns the character (of type `char`) that was typed, while the second returns a code of type `int` representing the character. The second of these often turns out to be the most useful in that not all keys (e.g., the arrow keys) return distinct characters.

The class `KeyEvent` contains integer constants, called virtual key codes, for each of the keys on the keyboard. For example, keys corresponding to letters on the keyboard are associated with constants `VK_A` through `VK_Z`, digits are associated with `VK_0` to `VK_9`, and the arrow keys are associated with `VK_UP`, `VK_DOWN`, `VK_LEFT`, and `VK_RIGHT`. Constants associated with other keys can be found in the on-line documentation for the class `KeyEvent`, which is in the package `java.awt.event`.

Sample code that uses arrow keys to change the speed of a falling ball is shown in Figure 10.17. When the up arrow key is pressed the ball's speed increases, while the down arrow key reduces the speed (and can even make it negative). The methods `keyTyped` and `keyReleased` must be included, even though their bodies are empty. They must appear because the listener must implement all of the methods of `KeyListener`, even though we only want to associate an action with `keyPressed` events.

10.7.2 Mouse Events

You have been reading and writing code to handle events generated by mouse actions from the beginning of this text, but you have been using features of the objectdraw package to simplify the code to be written. Not surprisingly, Java also provides a way to handle mouse events.

We explain how standard Java handles these events in detail in Appendix **??**, so we content ourselves with explaining only the basic ideas here. Fundamentally, mouse events are handled very similarly to keyboard events. You associate an appropriate mouse listener with the canvas and then write appropriate methods to handle the mouse events.

Mouse events can be generated by mouse clicks, presses, and releases, by having the mouse enter or leave a window, and by moving or dragging the mouse. The methods `mouseClicked`, `mousePressed`, `mouseReleased`, `mouseEntered`, and `mouseExited`, which handle the first five kinds of events are part of the interface `MouseListener`, while the methods `mouseMoved` and `mouseDragged` for handling moving and dragging events are part of the interface `MouseMotionListener`. As with classes implementing `KeyListener`, if a class implements one of the mouse listener interfaces then it must provide methods for all of the associated methods, even if they are not needed in the class.

All of the mouse event-handling methods take a parameter of type `MouseEvent` that contains information about where the event occurred. If `evt` is a `MouseEvent` then `evt.getX()` and `evt.getY()` return the x and y coordinates of where the mouse was when the event occurred.

Classes extending `WindowController` support methods with similar, but slightly different, names corresponding to each of these methods. Our methods also differ by taking a `Location`

```
import objectdraw.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class KeyBallController extends WindowController implements KeyListener {
    // Constants omitted
    private FallingBall droppedBall; //  the falling ball
    private int speed = SLOW_SPEED;  // speed of ball
    private JLabel speedLabel;       // label showing speed

    // create and install GUI items
    public void begin() {
        JPanel southPanel = new JPanel();
        speedLabel = new JLabel( "The speed is "+speed, JLabel.CENTER );
        southPanel.add( speedLabel );

        Container contentPane = getContentPane();
        contentPane.add( southPanel, BorderLayout.SOUTH );
        contentPane.validate();

        canvas.addKeyListener ( this );
        this.addKeyListener ( this );
requestFocusInWindow();
    }

    // make a new ball when the player clicks
    public void onMouseClick( Location point ) {
        droppedBall = new FallingBall( point, speed, canvas );
    }

    // Required by KeyListener Interface but not used here.
    public void keyTyped( KeyEvent e ) { }

    // Required by KeyListener Interface but not used here.
    public void keyReleased( KeyEvent e ) { }

    // Change speed with up and down arrow keys
    public void keyPressed( KeyEvent e )
    {
        if ( e.getKeyCode() == KeyEvent.VK_UP ) {
            speed++;
        } else if ( e.getKeyCode() == KeyEvent.VK_DOWN ) {
            speed--;
        }
        if ( droppedBall != null ) {
            droppedBall.setSpeed( speed );
        }
        speedLabel.setText( "The speed is "+speed );
    }
}
```

Figure 10.17: Controlling ball speed with the keyboard

as a parameter rather than an event. That was done in order to make it simpler for you to get the information you needed about each kind of mouse event.

The appendix explains how our methods are actually called by the standard Java mouse methods. Because of this interaction, you should avoid using standard Java mouse event-handling code with a class that extends `WindowController`.

10.8 Summary

In this chapter we introduced the standard Java event model and a number of GUI components. While there is a lot more that can be learned about the design of graphic user interfaces, the information provided here should be enough to get you started in writing programs using simple GUI components.

The tasks to be performed in creating and using GUI components include the following:

1. **Create the GUI component**.
2. **Add the GUI component to a container (either a panel or the content pane of a window) and validate it**.
3. **If a listener is needed,**
   (a) **Add `this` as a listener for the GUI component**.
   (b) **Add a declaration that the `WindowController` extension implements the appropriate listener interface**.
   (c) **Add to the `WindowController` the event-handling method promised by the listener interface**.

Tables 10.1 and 10.2 summarize the information needed to use GUI components and handle the associated events. Listeners for particular events are associated by sending a message of the form `add`*`SomeListener`* to the GUI component generating the event, where *`SomeListener`* should be replaced by the name of the listener. For example, to associate a listener with a button, send an `addActionListener` message to the button because the listener associated with buttons is `ActionListener`.

A `getSource` message can be sent to any event to obtain the object that generated the event. Events from the class `ChangeEvent`, generated by sliders, support `getValue` messages. Events from the class `KeyEvent` support methods `getKeyCode` and `getKeyChar`, while events from `MouseEvent` support methods `getX` and `getY` that return integer x and y coordinates of where the mouse was when the event was generated.

We discussed three Java layout managers for container classes like windows and panels. The `FlowLayout` manager inserts GUI components in the container from left to right, starting a new row when there is no more space available. It is the default layout for panels. `GridLayout` divides the container into a rectangular grid, where each cell has the same size. It inserts components from left to right starting with the top row, proceeding to the next row after each row is filled. `BorderLayout` is the default for content panes of extensions of `WindowController`. That layout manager allows items to be inserted in the center, north, south, east, and west of the container. The layout manager for a container may be changed by sending it the message `setLayout( manager )` where `manager` is a new object generated from one of the layout managers.

Component	Constructors	Associated Methods
JButton	new JButton(String label)	String getText() void setText(String newText)
JComboBox	new JComboBox()	void addItem(Object item) void setSelectedItem(Object item) Object getSelectedItem() void setSelectedIndex(int itemNum) int getSelectedIndex()
JSlider	new JSlider(int orientation, int lowVal, int highVal) int startValue)	int getValue()
JLabel	new JLabel(String text)	String getText() void setText(String newText)
JTextField	new JTextField(String initialContents, int numCols) new JTextField(String initialContents) new JTextField(int numCols)	String getText() void setText(String newText)
JTextArea	new JTextArea(String initialContents, int numRows, int numCols)	String getText() void setText(String newText) void append(String newText)

Table 10.1: Constructors and methods associated with GUI components

Event type	Generated by	Listener Interface	Method invoked
ActionEvent	JButton, JTextField	ActionListener	void actionPerformed(evt)
ItemEvent	JComboBox	ActionListener	void actionPerformed(evt)
ChangeEvent	JSlider	ChangeListener	void stateChanged(evt)
KeyEvent	Keyboard	KeyListener	void keyTyped(evt) void keyPressed(evt) void keyReleased(evt)
MouseEvent	Mouse	MouseListener	void mouseClicked(evt) void mousePressed(evt) void mouseReleased(evt) void mouseEntered(evt) void mouseExited(evt)
MouseEvent	Mouse	MouseMotionListener	void mouseDragged(evt) void mouseMoved(evt)

Table 10.2: Events, GUI components generating them, and associated interfaces and methods handling the events. Parameter `evt` always has the associated event type.

The `add` message is sent to a container in order to add a new component. With `FlowLayout` and `GridLayout` only the component to be added is sent as a parameter. However, with `BorderLayout` the location must also be specified. Thus a second parameter, one of `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST`, `BorderLayout.WEST`, or `BorderLayout.CENTER`, must be included.

You will discover that it often takes a fair amount of experimentation to make graphical user interfaces look good. Further information on Java GUI components from either the AWT or Swing package should be available on-line either locally or from `http://java.sun.com/apis.html`. Many books are also available on effective graphical user design. Our goal in this chapter was not to make you an expert on GUI design, but instead to introduce you to the standard GUI components and the use of standard Java event-driven programming.