Chapter 17

Introduction to Graphical User Interface Programming

This chapter explains how to build a graphical user interface (GUI) for an object-oriented program. In Chapter 4, the three tiers of a typical object-oriented program were presented. These comprise *Business Services*, *Data Services*, and *User Services*. The Business Services tier has been discussed throughout the book. Three ways to implement the Data Services tier were discussed in the fifth programming project. In the first two programming projects, the User Services tier was written as a text-based user interface that interacts with the objects in the Business Services tier. In the third, fourth and fifth programming projects, the User Services tier consisted of a set of GUI components; however, apart from a UML diagram in Chapter 11, no explanations were given on how to develop the GUI. Now, in order to facilitate a complete understanding of all three tiers of an object-oriented program, this chapter describes how to develop the User Services tier using a small set of GUI components.

This chapter is essentially a simplification of the knowledge and skills required to develop a GUI for an object-oriented program. It only touches the surface of the large set of GUI components that can be used to build the User Services tier of a Java application. At the end of the chapter, readers will understand how to develop the user interface code that will produce a GUI that is almost identical to the one used in the third, fourth and fifth programming projects. The chapter reinforces many of the object-oriented concepts covered in this book by describing the object-oriented nature of the GUI components in Java. Once readers are familiar with the basic concepts of GUI programming, there are many good resources that can be consulted for building more sophisticated GUIs.

17.1 The Swing Toolkit

The Swing Toolkit can be used for building graphical user interfaces and adding interactivity to Java programs. It provides a wide variety of components such as labels, text fields, buttons, check boxes and list boxes. To use components from the Swing Toolkit, the following `import` statement is required:

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

In addition to the Swing components, it is often necessary to use packages from the Java Abstract Windowing Toolkit (AWT) to provide functionality related to the GUI. The following `import` statements are usually required to use these packages:

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

The remainder of this chapter shows how to build an increasingly complex GUI using a minimal set of Swing components. Detailed explanations of the Swing components will not be provided since the objective of this chapter is to focus on the object-oriented aspects of the GUI. The Swing and AWT APIs can be consulted to obtain more detailed explanations of the GUI components mentioned in this chapter. The book Web site also contains instructions for using the Java API from a local computer or from the Internet.

17.2 An Empty Window

The first step in building a GUI is to create a top-level window. This section explains how to write the code to generate a top-level window. Two versions of the code are presented.

17.2.1 First Version

A fundamental concept in GUI programming is that of a *window*. A window is usually a rectangular portion of the monitor screen that can operate independently of the rest of the screen. A window has a width and a height. It also has a title and a border. `JFrame` is a Swing component that is used for creating top-level windows. To create a top-level window for a Java GUI, a class can be written that inherits from `JFrame`.

The first application we will look at in this chapter is `EmptyWindow`. This application creates a window and does nothing else. The window does not

contain any other components. A UML diagram of the classes in the application is given in Figure 17.1.

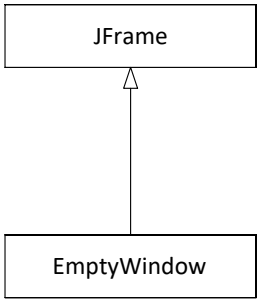

Figure 17.1: Classes in `EmptyWindow` Application

The `EmptyWindow` class inherits from `JFrame` and invokes some of its methods. It is written as follows:

```
import javax.swing.*;

public class EmptyWindow extends JFrame
{
   public EmptyWindow() {
      setTitle("An Empty Window");
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      setSize(300, 150);
      setVisible(true);
   }
}
```

The methods called in the constructor of `EmptyWindow` are either present in the `JFrame` class or are inherited from one of its ancestors (e.g., `Container` and `Component`). Since these methods are inherited, they can be used like any other method belonging to the `EmptyWindow` class. The methods called in the constructor of `EmptyWindow` are described in Table 17.1.

Method	*Description*
`setTitle()`	Puts a value on the title bar of the window.
`setSize()`	Establishes the dimensions of the window (width by height).
`setVisible()`	Makes the window disappear or reappear.
`setDefaultCloseOperation()`	Tells the application what to do when the user clicks on the "*Close*" button.

Table 17.1: Methods Used in `EmptyWindow`

The first argument of the `setSize()` method is the width of the window in pixels and the second argument is the height of the window in pixels. A newly created window is not visible on the desktop; to make the window visible, the `setVisible()` method must be used with an argument of `true`. Finally, the `setDefaultCloseOperation()` method tells the application what to do when the user clicks on the "*Close*" button at the top right hand corner of the window. The `EXIT_ON_CLOSE` argument tells the application to exit when the user clicks on "*Close*".

To create an instance of `EmptyWindow`, a `main()` method is written as follows:

```
public static void main(String[] args) {
   JFrame window = new EmptyWindow();
               // EmptyWindow "is-a" JFrame, by inheritance
}
```

The `main()` method can be written in a separate class or in the `EmptyWindow` class itself. Figure 17.2 gives a labeled screenshot of the window that is displayed when an instance of `EmptyWindow` is created.

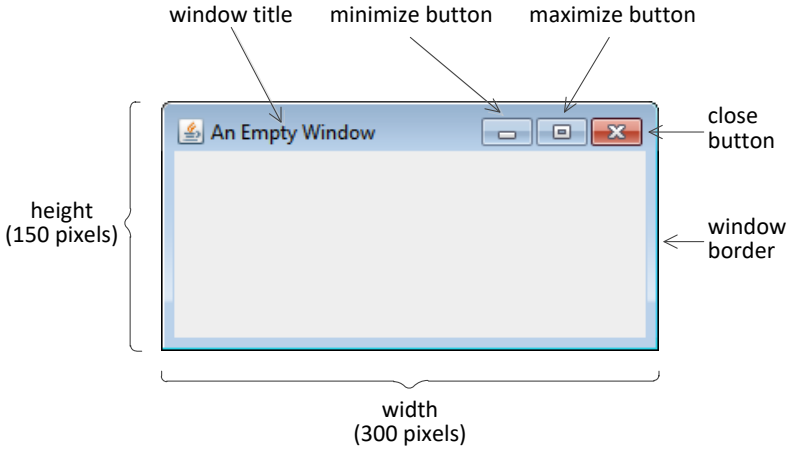

Figure 17.2: Window Displayed by `EmptyWindow`

Note that the window behaves just like the windows from other applications. For example, it can be re-sized by dragging the mouse at the borders. The window can be enlarged to occupy the entire desktop by clicking on the maximize button. The window can be made to temporarily "disappear" from the desktop by clicking on the minimize button. Finally, the application can be shut down by clicking on the close button.

17.2.2 Second Version

The simple `EmptyWindow` class can be enhanced by calling methods from its parent class `JFrame` or from the ancestors of `JFrame`. The Java API provides a detailed listing of the methods that are available in `JFrame` and its ancestor classes. Two enhancements to the constructor of `EmptyWindow` can easily be made. `EmptyWindow` presently positions the window at the top left hand corner of the screen. The `setLocationRelativeTo()` method of `Window` (an ancestor of `JFrame`) can be used to position the window at the center of the screen using an argument of `null`. Alternatively, the `setLocation()` method of `Component` can be used to position the window at some (`x`, `y`) location on the screen. Another change is to make it impossible for a user to adjust the size of the window by dragging on its borders. This is done by calling the `setResizable()` method of `Frame` (the direct superclass of `JFrame`) with a value of `false`.

The code for the enhanced constructor of `EmptyWindow` is given below:

```
public EmptyWindow() {
```

```
   setTitle("An Empty Window: Version 2");
   setDefaultCloseOperation(EXIT_ON_CLOSE);
   setSize(300, 150);

   setResizable(false);
                     // ensure that window cannot be resized

   setLocationRelativeTo(null);
                     // position window at center of screen

   setVisible(true);  // make the window visible
}
```

Now that we can display a window and manipulate the window in various ways, it is time to place some GUI components on the surface of the window. In the next section, we will learn how to do this using some simple GUI components.

17.3 Some Simple GUI Components

Consider the window shown in Figure 17.3. As the labels indicate, the surface of the window is populated with four types of visible GUI components: *labels*, *text fields*, *command buttons*, and a *text area*. This section describes each of these components. In the next section, it will be explained how these components can be put together to produce the GUI shown in Figure 17.3.

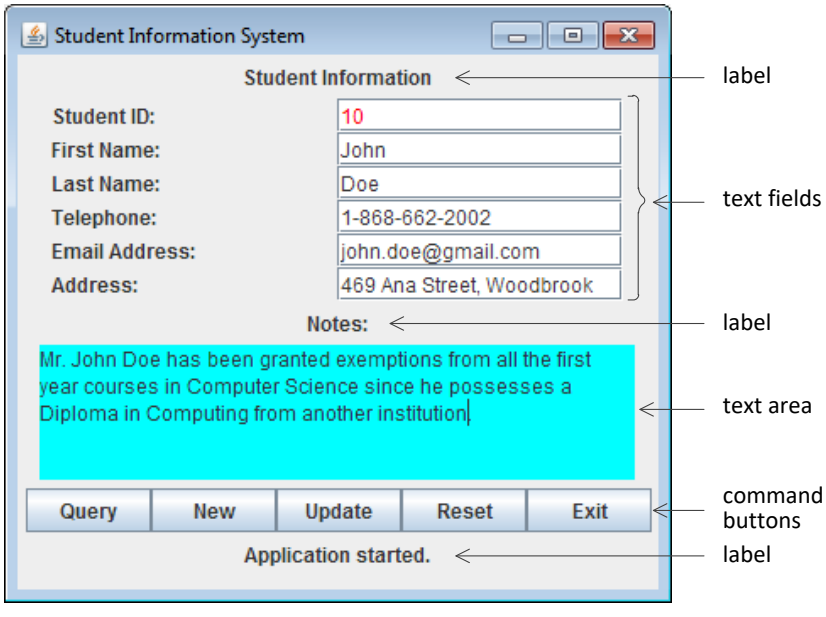

Figure 17.3: Window with Four Types of Visible GUI Components

17.3.1 Label

A *label* is a GUI component for displaying a short string or an image or both. It does not respond to input events such as clicking of the mouse or pressing a key on the keyboard. The Swing class corresponding to a label is `JLabel`. To create a label with the string "*Student Information*", an instance of `JLabel` is created as follows:

```
JLabel headingL;
headingL = new JLabel("Student Information");
```

A `JLabel` has methods such as `getText()` and `setText()` which can be used to retrieve or modify the label at any time. For example, to find out the textual label of a `JLabel` object, the following statement can be used:

```
String label = headingL.getText();
```

To change the text of the label, the following statements can be used:

```
String newLabel = "Student Personal Information";
headingL.setText(newLabel);
```

17.3.2 Text Field

A *text field* is a GUI component that can be used for editing a single line of text. Typically, it is used for data entry and data modification. The Swing class corresponding to a text field is `JTextField`. To create a text field that will enable a user to type the first name of a student, an instance of `JTextField` is declared and created as follows:

```
JTextField firstNameTF;
firstNameTF = new JTextField(15);
```

The argument 15 specifies the amount of columns that the text field should have. The amount of columns determines the width of the text field and this in turn determines the amount of characters that are visible. A `JTextField` will normally allow more characters to be typed than are visible. Like a `JLabel`, a `JTextField` has methods such as `getText()` and `setText()` which can be used to retrieve or modify the data in the text field at any time.

17.3.3 Text Area

A *text area* is a GUI component that facilitates the editing of several lines of text. It can be used in situations where the amount of characters to be typed by the user is difficult to predict. The Swing class corresponding to a text area is `JTextArea`. Arrow keys or the mouse can be used to position the cursor at any location inside the `JTextArea`.

To create a `JTextArea` that will enable a user to enter notes (or comments) on a student, a new empty `JTextArea` is declared and created as follows:

```
JTextArea notesTA;
notesTA = new JTextArea(5, 32);
```

The first argument of the constructor specifies the number of rows of the `JTextArea` and the second argument specifies the number of columns of the `JTextArea`.

Like a `JTextField`, a `JTextArea` has methods such as `getText()` and `setText()` which can be used to retrieve or modify the data in the `JTextArea` at any time.

17.3.4 Command Button

Command buttons are commonly used in GUI applications. They can be "pushed" by a user to perform a particular action. The Swing component for a

command button is `JButton`. A simple command button can be declared and created as follows:

```
JButton queryB;
queryB = new JButton("Query");
```

The string "*Query*" becomes the label that is displayed on the surface of the button. It can be modified at any time using the `setText()` method of `JButton`. Responding to a mouse click on the button is a little more complicated and will be discussed in a later section.

17.3.5 Layout Manager

GUI components must be placed at some location on a window (or other GUI container). In Java, a *layout manager* is responsible for positioning GUI components on a window or other container. There are several layout managers in Swing. In keeping with the object-oriented theme of the book, we are more interested in understanding the object-oriented nature of the GUI rather than the design and layout of the GUI. Thus, in this chapter, we will look at two of the simplest layout managers, `FlowLayout` and `GridLayout`.

`FlowLayout` is a very simple layout manager. It positions components in rows. If a component cannot fit on a row, it creates a new row and puts it there. When using `FlowLayout`, the width of a row is approximately the width of the window or container. GUI components are simply added to the container. The `FlowLayout` manager automatically positions the components by causing them to "flow" from one row to the next. If the user resizes the window or container, the `FlowLayout` manager re-positions the components based on the new width of the window or container. A `FlowLayout` manager is created as follows:

```
FlowLayout flowLayout = new FlowLayout();
```

`GridLayout` is another simple layout manager which uses a rectangular grid to position GUI components. A grid consists of a number of rows and columns, like a table. The rows and columns are specified when the `GridLayout` manager is created. The following statement creates a `GridLayout` manager for a grid that is 6 rows by 2 columns.

```
GridLayout gridLayout = new GridLayout(6, 2);  // 6 rows x 2 columns
```

The `GridLayout` manager positions the GUI components on the cells of the grid starting from the cell at the first row and first column. Figure 17.4 shows how components are positioned by the `GridLayout` manager; the numbers in

the cells indicate the order in which components are placed on the grid (starting from 1 and ending at 12):

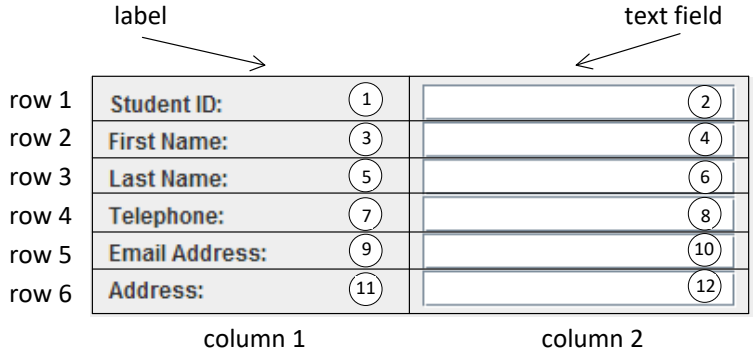

Figure 17.4: Placement of Components by `GridLayout` Manager

After creating an instance of the layout manager, it must be assigned to a window or container using the `setLayout()` method of the window or container. For example, to attach the `GridLayout` manager above to a window, the following statement is used:

```
setLayout(gridLayout);
```

17.4 StudentWindow

This section describes the code which generates the GUI shown in Figure 17.3. The GUI is used to enter, query, and update information on students. It uses the four visible GUI components discussed in the previous section. Two versions of the code are described. The first version uses the `FlowLayout` manager and the second version uses the `GridLayout` manager.

17.4.1 First Version of StudentWindow

Figure 17.5 shows the GUI generated by the first version of the code. A UML diagram of the components that generate the GUI is given in Figure 17.6. The UML diagram uses a different kind of labeling from what we have seen so far. Some of the rectangles contain a label which is underlined. The label consists of a class name prefixed with a colon, e.g., :JLabel, :JTextArea. This notation is used to represent an instance of a class rather than the class itself. For example, :JLabel inside a rectangle represents a single instance of the class, `JLabel`. If more instances are required, this is specified at the end of the relationship line. For example, the "5" at the end of the relationship line which connects

`StudentWindow` to :JButton indicates that `StudentWindow` contains five instances of `JButton`.

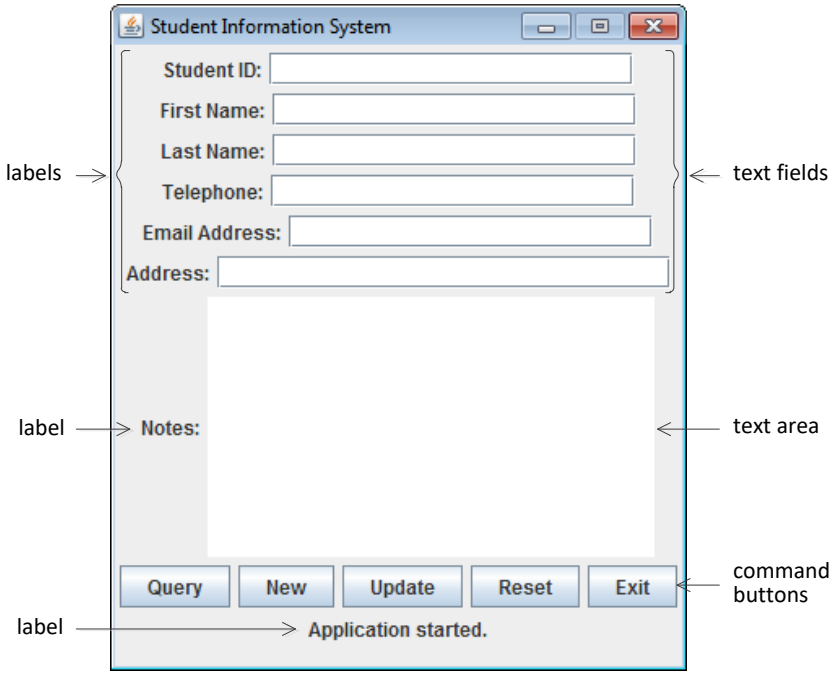

Figure 17.5 First Version of `StudentWindow` (using `FlowLayout`)

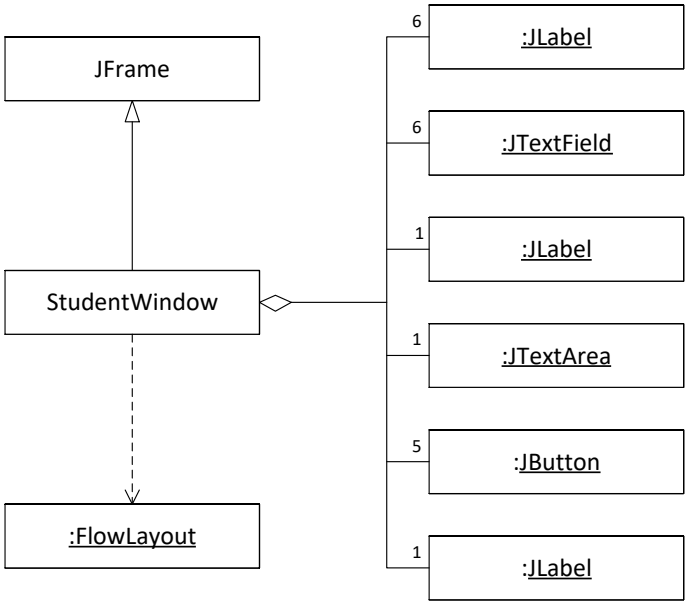

Figure 17.6: GUI Components in First Version of `StudentWindow`

The UML diagram shows that the `StudentWindow` class inherits from `JFrame`. `StudentWindow` contains six labels, six text fields, one label and a text area, five command buttons, and another label. `StudentWindow` uses an instance of the `FlowLayout` manager to position the GUI components on the window. The GUI components are positioned in the order shown in Figure 17.6, from top to bottom, except for the six labels and six text fields which are positioned in pairs.

All the GUI components are declared as instance variables of the `StudentWindow` class. In the constructor of `StudentWindow`, the components are created as discussed in the previous section. An instance of the `FlowLayout` manager is created and assigned to the window. The GUI components are simply added to the window in the order desired using the `add()` method of `JFrame`:

```
add(IDL);              // add ID label to the window
add(IDTF);             // add ID text field to the window

// repeat above code for next five labels and text fields

add(notesL);           // add label for text area
```

```
add(notesTA);          // add text area

add(queryB);           // add query command button

// repeat above code for next four command buttons

add(statusBarL);       // add status bar label
statusBarL.setText("Application started.");
                       // display message in status bar
```

Apart from the basic windowing statements discussed in Section 17.2, nothing more is required to generate the GUI shown in Figure 17.5. The complete code is available at the book Web site.

The `FlowLayout` manager produces a simple layout; however, it is generally not neat or attractive. The `GridLayout` manager can be used to produce a neater layout than the one shown in Figure 17.5. It is discussed in the next sub-section.

17.4.2 Second Version of `StudentWindow`

In this sub-section, the `StudentWindow` from the previous sub-section is enhanced so that it will generate the GUI shown in Figure 17.7. Incidentally, this is the same GUI shown in Figure 17.3. A UML diagram of the components that are used to generate the second version of the GUI is shown in Figure 17.8.

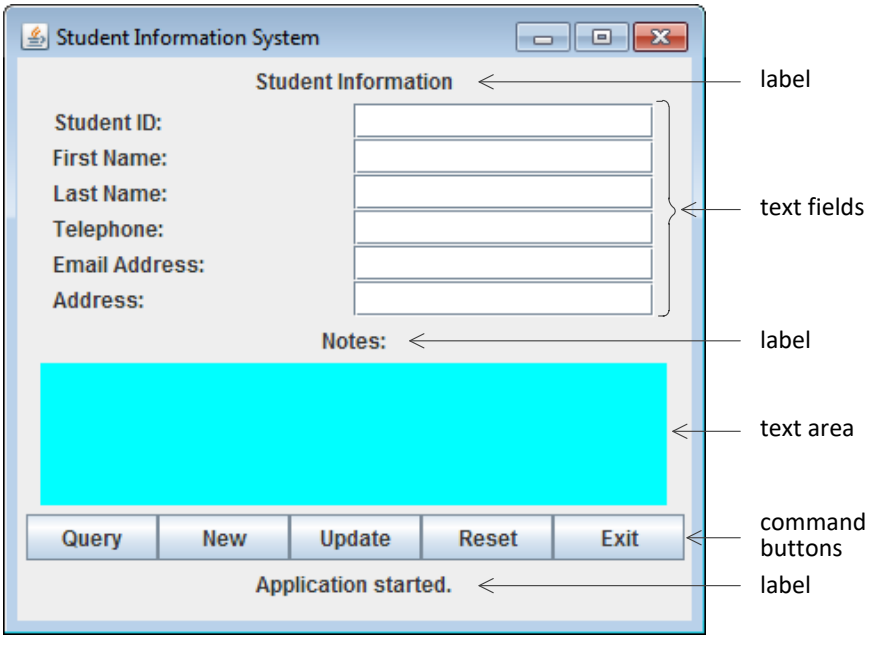

Figure 17.7 Second Version of `StudentWindow` (using `GridLayout`)

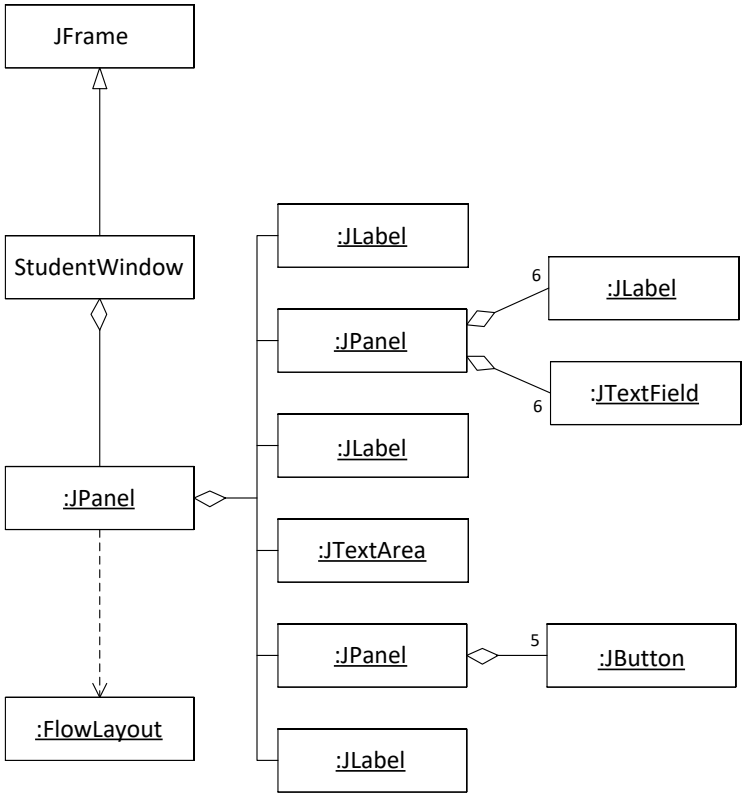

Figure 17.8: GUI Components in Second Version of `StudentWindow`

The two major differences in this version of the GUI are:

(1) The `GridLayout` manager is used to arrange the labels and text fields to give a neater appearance of the GUI components on the window, and

(2) Two panels are used to group GUI components together. The first panel groups together the six labels and six text fields at the top of the window. The second panel groups together the five command buttons.

A *panel* is a container that can contain other GUI components. Each panel can use its own layout manager. The Swing class corresponding to a panel is `JPanel`. The first `JPanel` uses the `GridLayout` manager; the dimensions of the grid are 6 rows by 2 columns (see Figure 17.4). Each row contains a label

followed by a text field. The second `JPanel` also uses the `GridLayout` manager; the dimensions of the grid are 1 row by 5 columns. This causes the command buttons to be neatly displayed in one row.

The following code shows how the two `JPanel`s are created and how the layout managers are created and assigned to each one:

```
GridLayout gridLayout;

JPanel topPanel = new JPanel();
                              // JPanel for labels/textfields at top

gridLayout = new GridLayout(6, 2);
                              // grid is 6 rows x 2 columns

topPanel.setLayout(gridLayout);
                              // assign layout manager to JPanel
:

JPanel buttonPanel = new JPanel();
                              // JPanel for buttons at bottom

gridLayout = new GridLayout(1, 5);
                              // grid is 1 row x 5 columns

buttonPanel.setLayout(gridLayout);
                              // assign layout manager to JPanel
```

After creating the `JPanel`s, the GUI components are added to each `JPanel` using the `add()` method of `JPanel`.

As shown in Figure 17.8, all the GUI components are themselves grouped together in another `JPanel` which is attached to the `StudentWindow`. This `JPanel` uses a `FlowLayout` manager to position the sub-panels, labels, and text area on the surface of the panel.

17.5 Three Advanced GUI Components

This section explains how to use three advanced GUI components: *combo boxes*, *radio buttons*, and *check boxes*. These GUI components make it easier for the user to enter data and reduce the risk of data-entry errors. Each component is explained in the context of creating and updating a `Student` object and displaying data from an existing `Student` object. The section concludes by

showing how to enhance the `StudentWindow` from the previous section with these three components.

17.5.1 Combo Box

A *combo box* combines a button with a drop-down list. If the user clicks on the button, a drop-down list is displayed. The user can scroll down the drop-down list and select a value which is then displayed. Figure 17.9 shows a combo box which allows a user to choose from a list of countries.

Figure 17.9: Combo Box with List of Countries

The functionality of a combo box is provided in Swing by `JComboBox`. A `JComboBox` must be supplied with the set of items to populate the drop-down list (this is referred to as the *data model*). The data model can be supplied as an array of `Object`s when the `JComboBox` is created; the `Object`s in the array must implement the `toString()` method. For example, to create a `JComboBox` to display the list of five countries shown in Figure 17.9, the following code can be used:

```
String[] countries = new String
   {"Barbados", "Grenada", "Guyana", "Jamaica",
   "Trinidad & Tobago"};

JComboBox countriesCB = new JComboBox(countries);
```

Alternatively, the `JComboBox` can be created with an empty data model and then the items for the data model can be supplied using its `addItem()` method:

```
JComboBox countriesCB = new JComboBox();
countriesCB.addItem("Barbados");
countriesCB.addItem("Grenada");
countriesCB.addItem("Guyana");
countriesCB.addItem("Jamaica");
```

```
countriesCB.addItem("Trinidad & Tobago");
```

To automatically position the combo box at a particular country, the index of that country can be specified as the argument of the `setSelectedIndex()` method. For example, to position the combo box on "*Jamaica*" when it is initially displayed, the following code can be used:

```
countriesCB.setSelectedIndex(3);            // index of Jamaica is 3
```

Alternatively, the name of the country can be specified as the argument of the `setSelectedItem()` method of the `JComboBox`. For example,

```
countriesCB.setSelectedItem("Jamaica");     // use name of country
```

If the index of the country is not within the range of valid indices of the data model (or -1), an `IllegalArgumentException` is thrown when the `setSelectedIndex()` method is called. On the other hand, if the name of the country supplied as an argument to the `setSelectedItem()` method is not present in the data model, the combo box is positioned at the first item in the data model.

To create a new `Student` object (or update an existing one), we must find out which of the items in the drop-down list is currently selected by the user. To do so, the `getSelectedItem()` method can be used:

```
String country = countriesCB.getSelectedItem().toString();
```

The `getSelectedItem()` method returns an `Object` so the `toString()` method should be used to get a string representation of the object displayed. The `country` string is then stored as an attribute of the new or updated `Student` object.

To display the `country` attribute of an existing `Student` object, the combo box must position the drop-down list at the given country. To do this, the data model of the `JComboBox` must be searched to find the index of the country in its list of countries. The index obtained is used with the `setSelectedIndex()` method to cause the combo box to display the given country:

```
String country = student.getCountry();
        // obtain value of country attribute from Student object

int countryIndex = getCountryIndex(country);
        // obtain index corresponding to country
```

```
if (countryIndex >= 0)
        // index is valid

   countriesCB.setSelectedIndex(countryIndex);
        // combo box displays country at given index
```

Note that `getCountryIndex()` is a method which searches the data model of the `JComboBox` to find the index of the given country.

17.5.2 Radio Button

A *radio button* is a GUI component that can be selected or de-selected by the user. It is available in Swing as `JRadioButton`. A `ButtonGroup` object can be used to group together a set of `JRadioButton` objects so that only one `JRadioButton` at a time can be selected.

Figure 17.10 shows a set of `JRadioButton` objects which allow the user to specify the attendance `status` of a student:

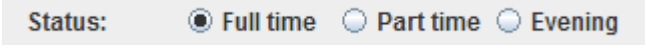

Figure 17.10: Radio Buttons for Attendance `Status`

To create the `JRadioButton` objects, the following code can be used:

```
// declare radio buttons

JRadioButton status1, status2, status3;
ButtonGroup statusGroup;        // to group radio buttons together

// create radio buttons

status1 = new JRadioButton("Full time");
status2 = new JRadioButton("Part time");
status3 = new JRadioButton("Evening");
statusGroup = new ButtonGroup();// create instance of ButtonGroup
```

The `JRadioButton` objects are then grouped together by the `ButtonGroup` object:

```
statusGroup.add(status1);
statusGroup.add(status2);
statusGroup.add(status3);
```

Once the three `JRadioButton` objects are added to the `ButtonGroup`, only one at a time can be selected. If they are not added to the `ButtonGroup`, any combination of the three can be selected.

When displayed for the first-time, all the `JRadioButton` objects are de-selected. To automatically select one of the `JRadioButton` objects, the `setSelected()` method can be used. For example, since "*Full time*" students are the most common, the "*Full time*" `JRadioButton` can be pre-selected with the following code:

```
status1.setSelected(true);
```

To find out at any time if a `JRadioButton` is selected or de-selected, its `isSelected()` method can be invoked. For example, the following code can be used to determine which of the three radio buttons is selected:

```
String status;
if (status1.isSelected())
   status = status1.getText();
else
if (status2.isSelected())
   status = status2.getText();
else
if (status3.isSelected())
   status = status3.getText();
```

At the end of the `if` statement, the `status` string will contain the label of the `JRadioButton` that is currently selected. The label of a `JRadioButton` is the string that was supplied to the constructor (e.g., "*Part time*"). The `status` string can then be used to update the `status` attribute of a `Student` object.

To display the `status` attribute of a given `Student` object on the GUI, it is necessary to select the appropriate `JRadioButton` from the `ButtonGroup`. The following code can be used for this purpose:

```
String status = student.getStatus();
if (status1.getText().equals(status))
   status1.setSelected(true);
else
if (status2.getText().equals(status))
   status2.setSelected(true);
else
if (status3.getText().equals(status))
   status3.setSelected(true);
```

The code checks to see which of the `JRadioButton`s corresponds to the `status` attribute. The `JRadioButton` with the same label as the `status` attribute is selected through its `setSelected()` method.

17.5.3 Check Box

A *check box* is similar to a radio button and can be selected or de-selected by the user. A check mark is usually placed inside the check box to indicate that it has been selected. If a group of check boxes is used, the user can select as many of them as required. The Swing component corresponding to a check box is `JCheckBox`. In the example Student Information System, six check boxes are used to specify the extra-curricular activities a student is engaged in. They are shown in Figure 6.11.

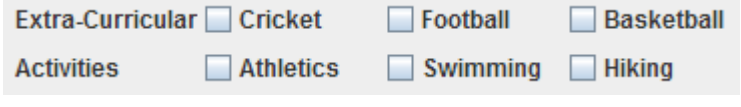

Figure 6.11: Check Boxes for Extra-Curricular Activities

Check boxes can be created and manipulated individually like radio buttons. However, when there are many check boxes, the coding becomes tedious so it is more effective to create an array of check boxes. So, the six check boxes in the Student Information System can be declared and created as follows:

```
JCheckBox[] ecActivities = new JCheckBox[6];
                      // array of 6 check boxes

ecActivities[0] = new JCheckBox("Cricket");
ecActivities[1] = new JCheckBox("Football");
ecActivities[2] = new JCheckBox("Basketball");
ecActivities[3] = new JCheckBox("Athletics");
ecActivities[4] = new JCheckBox("Swimming");
ecActivities[5] = new JCheckBox("Hiking");
```

Unlike a combo box or `ButtonGroup` of radio buttons, a container rather than a single variable is needed to store the currently selected set of check boxes. Assuming that an `ArrayList` is used, the code to store the selections is as follows:

```
ArrayList<String> ecActivitiesSelected = new ArrayList<String>();
for (int i=0; i<ecActivities.length; i++) {
           // examine each check box
```

```
   if (ecActivities[i].isSelected())  {
            // is check box selected?

      String label = ecActivities[i].getText();
            // get label of check box

      ecActivitiesSelected.add(label);
            // store label in ArrayList
   }
}
```

The code examines each check box to determine if it is selected. If so, the label of the checkbox is added to the `ArrayList` of selections. After the loop exits, the `ArrayList` of selections is used to update the corresponding attribute in a new or existing `Student` object. Note that the `ArrayList` stores the string that was used in the constructor of the `JCheckBox`. However, it might be more efficient to store indexes rather than an entire string.

To display the `ArrayList` of selections from a given `Student` object, the reverse process occurs. For each `JCheckBox` in the `ecActivities` array, we must check to see if its label is stored in the `ArrayList` (which indicates that this `JCheckBox` was selected for the given `Student`). If so, the `setSelected()` method is used to automatically select that `JCheckBox`. The code is as follows:

```
for (int i=0; i<ecActivities.length; i++) {
   if (ecActivitiesSelected.contains(ecActivities[i].getText()))  {
      ecActivities[i].setSelected(true);
   }
}
```

17.5.4 Final Version of `StudentWindow`

We would now like to add the following GUI components to the `StudentWindow` shown in Figure 17.7:

- A combo box, to allow selection from a list of countries of the world
- Three radio buttons, to allow selection of one of the three different possibilities for attendance status
- Six check boxes, to allow selection of extra-curricular activities

The radio buttons and check boxes used in the final version of `StudentWindow` are exactly as described in the previous two sub-sections. However, the combo box must display a list of all the countries in the world

instead of only five countries. This is accomplished by storing a list of the countries in a text file, `Countries.txt`. As explained in Chapter 15, a `BufferedReader` can be used to read the text file, line by line. The countries are stored in an `ArrayList`. Next, the elements of the `ArrayList` are inserted into the combo box using the `addItem()` method of `JComboBox`.

The combo box, radio buttons, and check boxes are inserted at the appropriate positions in the GUI. Figure 17.12 is a screenshot of the actual window generated. Figure 17.13 is a UML diagram of the classes and objects in the final version of `StudentWindow`.

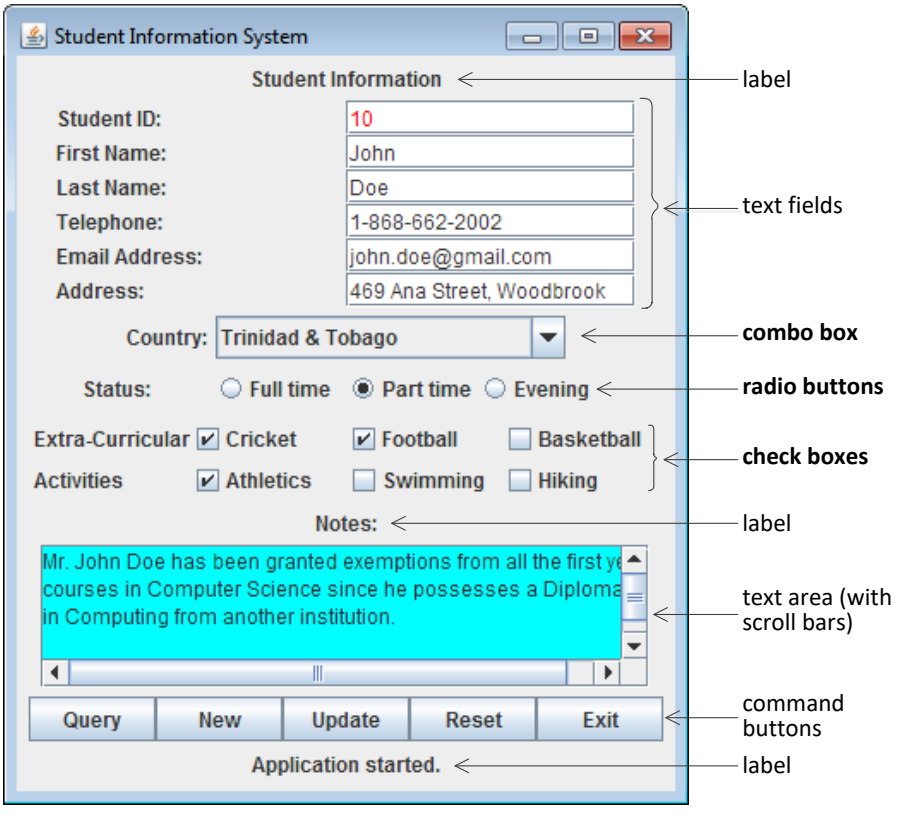

Figure 17.12: Screenshot of Final Version of `StudentWindow`

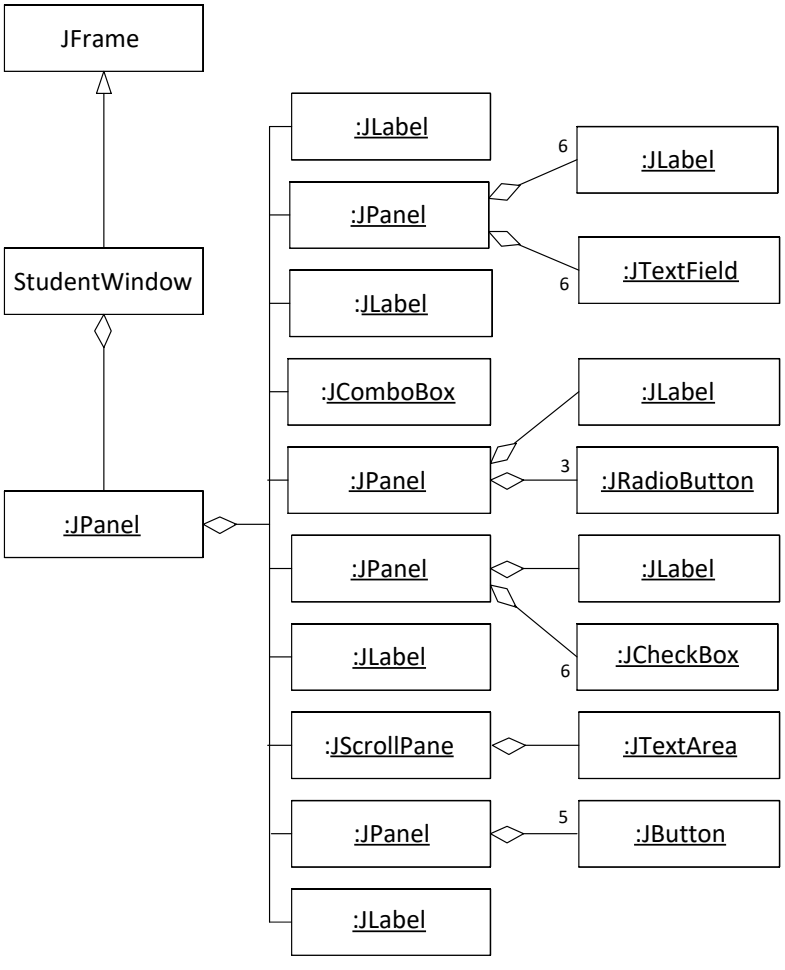

Figure 17.13: Classes and Objects in Final Version of `StudentWindow`

It can be observed that the *Notes* text area looks a little different from the previous version. There seems to be a border around the text area; also, if text is typed in the text area and it goes beyond the width or height of the text area, scroll bars automatically appear. The scroll bars operate just like the scroll bars in other applications. To get scroll bars around the text area, a `JScrollPane` must be used. The following is the code required to use a `JScrollPane`:

```
JScrollPane notesScroller;
notesScroller = new JScrollPane();
                // create instance of JScrollPane
```

```
notesScroller.getViewport().add(notesTA);
                // insert text area in JScrollPane
```

Note that the scroll bars only appear when the text goes beyond the viewable area of the text area.

At this point in time, the GUI is exactly as we want it in terms of its physical appearance. However, if we click on any of the command buttons, nothing happens. Clicking on the combo boxes and other GUI components causes some action to take place that is specific to the GUI component. However, no application-specific behavior takes place. In the next section, we will learn how to get the GUI to respond to events such as clicking of the mouse or pressing *Enter* on the command buttons.

17.6 Responding to Events on the `StudentWindow`

Command buttons and other user interface components are built-in components that provide a wide variety of functionality depending on the requirements of a given application. A very important issue in developing a graphical user interface is how to respond to an event such as a click on a command button. This is known as *event-driven programming*. Event-driven programming involves writing application-specific code to take some action when a pre-determined event occurs. This code is referred to as an *event handler*. This section describes the event handling mechanism in Java and explains how three main types of events are handled.

17.6.1 Event Handling Mechanism

In writing event handling code, a basic problem is how to connect application-specific code to a built-in GUI component that has no prior knowledge of the application-specific code. This problem is solved in Java by using the concept of an interface. Recall that an interface makes it possible to deal with objects which are not known except that they implement a minimum set of behaviors specified by the interface. In order to respond to an event occurring in a built-in GUI component, the class that contains the event-handling code must implement a specific interface defined by the designers of the GUI component. Table 17.2 lists the interfaces for some common events on a GUI component (there are many more).

Event	*Interface*
Clicking on a command button or pressing the Enter key on a command button	`ActionListener`
Pressing a key on the keyboard	`KeyListener`
Clicking the mouse on the window surface	`MouseListener`

Table 17.2: Common Events on a GUI

The event handler for a particular event must be written in a class that implements the appropriate interface. Indeed, the event handlers *are* the methods implemented from the interface together with any other helper methods. For example, to specify what to do when a user clicks on a command button, a class must be written that implements the *ActionListener* interface. Table 17.3 lists the methods that must be implemented for each of the interfaces given in Table 17.2.

Interface	*Methods of Interface*	*Event Class*
ActionListener	**void actionPerformed(ActionEvent e)**	ActionEvent
KeyListener	**void keyPressed(KeyEvent e)** void keyReleased(KeyEvent e) void keyTyped(KeyEvent e)	KeyEvent
MouseListener	**void mouseClicked(MouseEvent e)** void mouseEntered(MouseEvent e) void mouseExited(MouseEvent e) void mousePressed(MouseEvent e) void mouseReleased(MouseEvent e)	MouseEvent

Table 17.3: Event Handling Methods

Each of the three interfaces in Table 17.3 has an associated `Event` class. An instance of this `Event` class is passed as an argument to the method/s implementing the interface. The `Event` object provides detailed information about the event that occurred. For example, an instance of `ActionEvent` is passed as an argument to the `actionPerformed()` method. It provides information such as the title of the command button which generated the event.

Only the three methods highlighted in bold in Table 17.3 will be discussed in this chapter. Empty method bodies will be provided for the remaining

methods since a concrete class implementing an interface must provide an implementation for all the methods declared in the interface.

It is convenient for the class that generates the window to implement the relevant interface itself. So, to respond to a button click on any of its buttons, the `StudentWindow` class can itself implement the `ActionListener` interface.

17.6.2 Clicking on a Command Button

In order to respond to a click on a command button, the `ActionListener` interface must be implemented. This interface has only one method, `actionPerformed()`. The implementation of the `actionPerformed()` method must specify what to do when the user clicks on one of the command buttons.

A simple example of the implementation of the `actionPerformed()` method is as follows:

```
public void actionPerformed(ActionEvent e) {
   String command = e.getActionCommand();
   statusBarL.setText(command + " button clicked.");
}
```

The `ActionEvent` object passed to the `actionPerformed()` method can be queried to find out more information about the event that occurred. For example, its `getActionCommand()` method returns the label of the button which the user clicked on. In the example above, the `actionPerformed()` method simply displays the label together with a short string on the status bar. Clicking or pressing *Enter* on any of the five buttons causes a message to be displayed on the status bar indicating which button was pressed.

The `actionPerformed()` method may need to refer to GUI components on the window. In the example above, it needs to refer to the status bar. In other cases, it may need to refer to the text fields and other GUI components on the window. For this reason, the GUI components should be declared as instance variables rather than local variables in the class implementing the `ActionListener` interface (in this case, `StudentWindow`).

17.6.3 Pressing a Key on the Keyboard

If we wanted to know when the user pressed a key on the keyboard, the `KeyListener` interface should be implemented. This interface has three

methods as shown in Table 17.3. A simple `keyPressed()` method can be written as follows:

```
public void keyPressed(KeyEvent e) {
   int keyCode = e.getKeyCode();
   String keyText = e.getKeyText(keyCode);
   statusBarL.setText(keyText + " pressed.");
}
```

Like the `ActionEvent` object passed to the `actionPerformed()` method, the `KeyEvent` object passed to the `keyPressed()` method can be queried to get more information on the keyboard event that occurred. In the example above, the `getKeyCode()` method of `KeyEvent` returns an integer code representing the key that was pressed. The `getKeyText()` method returns a string representation of the integer code such as "*A*", "*F1*", "*Escape*", etc. The string is displayed on the status bar.

Empty method bodies are written for the remaining methods in the interface. For example,

```
public void keyReleased(KeyEvent e) {

}
```

Of course, if you wish to take some action when a key is released, event handling code should be written for the `keyReleased()` method.

17.6.4 Clicking the Mouse

To find out if the user clicked the mouse somewhere on the window (in an area not occupied by a GUI component), the `MouseListener` interface should be implemented. There are five methods in the `MouseListener` interface. In the following example, only the `mouseClicked()` method is implemented. Empty method bodies are provided for the remaining four methods declared in the interface.

```
public void mouseClicked(MouseEvent e) {
   int x = e.getX();
   int y = e.getY();
   statusBarL.setText("Mouse click at (" + x +", " + y + ")");
}
```

In the above example, the `MouseEvent` object passed to the `mouseClicked()` method gives more information about the mouse click event. It can be queried

to find out the (`x`, `y`) coordinates of the point at which the mouse was clicked. After finding out the coordinates, the `mouseClicked()` event handler simply displays the coordinates on the status bar.

An important point to note about event handlers is that they are not called explicitly. Whenever an event occurs on a GUI component, the Java run-time system creates an instance of the appropriate event and passes it to an event handler connected to that GUI component (if one exists). For example, if a user clicks on a command button, the Java run-time system creates an instance of `ActionEvent` and passes it to the `actionPerformed()` method. If the `actionPerformed()` method is not implemented, the event will not be handled and nothing happens. This is exactly what happened in the previous versions of `StudentWindow`.

17.6.5 Attaching Event Handlers to GUI Components

Besides implementing the relevant `Listener` interface, there is one more task that must be done to complete the event handling process. This involves attaching the event handler to the appropriate GUI component or components. If this is not done, the event handler will not be activated by the Java run-time system when the event of interest occurs on the GUI component. Table 17.4 gives some examples of attaching an event handler to a GUI component for the three types of events listed in Table 17.2.

Event	*GUI Component*	*Declaration of Component*	*Code to attach event handler*
Mouse click or pressing *Enter* on a command button	JButton	JButton queryB; JButton newB;	queryB.addActionListener(this); newB.addActionListener(this);
Pressing a key on the keyboard	JTextArea	JTextArea notesTA;	notesTA.addKeyListener(this);
Clicking the mouse on the surface of a panel	JPanel	JPanel mainPanel;	mainPanel.addMouseListener(this).

Table 17.4: Attaching Event Handlers to GUI Components

The `this` argument of the `addListener()` methods listed in Table 17.4 indicates that the current class contains the event handler. If the event handler is implemented in another class, an instance of this class must be supplied to the `addListener()` methods in Table 17.4. Note that the same event handler can be attached to more than one GUI component. For example, all the command buttons can have the same event handler.

17.7 Communicating with Domain Objects in `StudentWindow`

In Chapter 4, the three-tier architecture of an object-oriented application was presented. It was mentioned that the User Services tier is relatively free of application processing and forwards task requests to the Business Services tier. In Chapter 16, it was shown how the User Services tier may also forward task requests related to persistence to the Data Services tier. An important question is this: how should the objects in the Business Services tier communicate with objects in the User Services tier? In other words, how should domain objects communicate with user interface objects?

17.7.1 Model-View Separation

The *Model-View Separation* design pattern states that model objects (i.e., domain objects or objects in the Business Services tier) should *not* have direct knowledge of or be directly coupled to view objects (i.e., the user interface objects) (Larman, 1997). Thus, a model class should not have knowledge or code related to user interfaces. In order for the objects in the view to display information, methods must be invoked on the relevant objects in the model. The data is then displayed via GUI components such as those we have already seen in this chapter. This approach is called the *pull-from-above* method. In support of this principle, none of the domain classes in this book contains code that directly communicates with the user interface (text-based or otherwise).

There are several advantages that are gained by using the Model-View Separation design pattern. For example,

- The model can be developed independently of the user interface
- New views can be easily connected to an existing model without affecting the objects in the model
- Multiple, simultaneous views of the same model can be developed (e.g., a desktop view, a smart phone view, a Web form view)
- The classes in the model can easily be ported to another user interface

17.7.2 Writing Code that Implements Pull-From-Above

The `StudentWindow` (the view) has buttons to enable the querying of students, inserting new students, and updating the information on students. The classes of the model (or the domain classes) are `Student` and `University`. A UML diagram of the model classes is given in Figure 17.14.

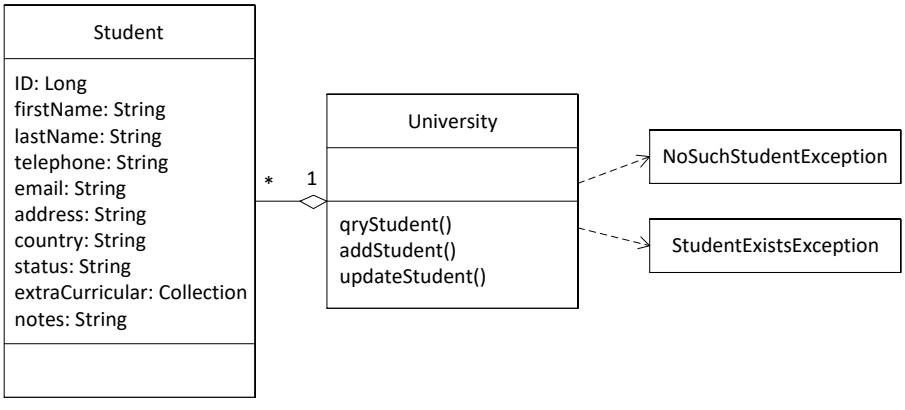

Figure 17.14: Model Classes Accessed by `StudentWindow`

Essentially, the view will use pull-from-above to communicate with the model classes. The event handler for the *Query* button will invoke the `qryStudent()` method on a `University` instance, supplying the value of the `ID` text field as an argument. Similarly, the event handler for the *New* button will invoke the `addStudent()` method on a `University` instance, supplying the values from the different GUI components which accept input as arguments. The event handler for the *Update* button will invoke the `updateStudent()` method on a `University` instance, again supplying the values from the different GUI components as arguments. The `addStudent()` method will throw an exception if a student with the given `ID` already exists. The `updateStudent()` method will also throw an exception if a student with the given `ID` cannot be found. In both cases, the view (i.e., the instance of `StudentWindow`) must catch the exception and do something with it.

The same `actionPerformed()` method handles events from all the command buttons. However, a helper method performs task-specific event handling for each command button. The `actionPerformed()` method determines which button was pushed and then calls the appropriate helper method:

```
public void actionPerformed(ActionEvent e) {

   String command = e.getActionCommand();

   if (command.equals(queryB.getText())) {  // Query button clicked
      queryStudent();
   }
   else
   if (command.equals(newB.getText())) {    // New button clicked
      newStudent();
```

```
    }
    else
    if (command.equals(updateB.getText())) { // Update button clicked
       updateStudent();
    }
    else
    if (command.equals(resetB.getText())) {  // Reset button clicked
       clearFields();
    }
    else
    if (command.equals(exitB.getText()))     // Exit button clicked
       System.exit(0);
}
```

The `queryStudent()`, `newStudent()`, and `updateStudent()` helper methods invoke methods on the `University` instance to get their work done. However, there is some additional work that must be done by the helper methods. This is described in Table 17.5.

Event handler helper method	*Additional work before calling relevant method of `University`*	*Additional work after calling relevant method of `University`*
queryStudent()	Obtains `studentID` from `ID` text field and ensures that it is not `null`. Converts `studentID` to a `long` value and invokes `qryStudent()` on the `University` instance.	If a valid `Student` object is returned, invokes accessor methods on the `Student` object to obtain data. This data is displayed on the GUI components.
addStudent(), updateStudent()	Obtains `studentID` from `ID` text field and ensures that it is not `null`. Obtains data from the GUI components that accept input and invokes `addStudent()` or `updateStudent()` on the `University` instance.	Displays error message if exception is generated.

Table 17.5: Work Done by Event Handler Helper Methods

The following code shows how the `newStudent()` helper method is implemented:

```
private void newStudent() {

   String ID = IDTF.getText();
                // obtain ID string

   if (ID.length() == 0) {
                // check if the string has anything in it

      statusBarL.setText("Student ID must be specified.");
      return;
   }

   try {
      Student student;
      String country = countriesCB.getSelectedItem().toString();
                // find which country is currently selected

      String status = null;

      if (status1.isSelected())
                // find which status button is selected

         status = status1.getText();
      else
      if (status2.isSelected())
         status = status2.getText();
      else
      if (status3.isSelected())
         status = status3.getText();

                // request University to create Student
      student = university.addStudent(Long.parseLong(Id),
                firstNameTF.getText(),
                lastNameTF.getText(),
                addressTF.getText(),
                country,
                telephoneTF.getText(),
                emailTF.getText(),
                status);

      ArrayList<String> ecActivitiesSelected =
         new ArrayList<String>();

      for (int i=0; i<ecActivities.length; i++) {
                // examine each check box
```

```
        if (ecActivities[i].isSelected()  {
                // is check box selected?

           String label = ecActivities[i].getText();
                // get label of check box

           ecActivitiesSelected.add(label);
                // store label in ArrayList
        }

        student.setExtraCurricular(ecActivitiesSelected);
                // set extraCurricular attribute

        student.setNotes(notesTA.getText());
                // set notes attribute

        statusBarL.setText("Student successfully added.");
                // display message on status bar

     }
   }
   catch (StudentExistsException see) {
                // catch exception

      statusBarL.setText
         ("Error: A student with this ID already exists.");
   }
}
```

At this stage, the `StudentWindow` has the appearance that we want and it can also respond to mouse clicks on the buttons in a meaningful way. However, a few enhancements will be made before the chapter concludes.

17.8 Enhancing the GUI

In this section, we will show how to enhance the GUI so that it will resemble the one used in the third, fourth, and fifth programming projects. First, a *tabbed pane* is used so that several views can be accommodated on the same window by clicking on a tab. Next, a *menu bar* is added to the window. Finally, several *popup windows* will be used to display different kinds of information.

17.8.1 Supporting Multiple Windows

A `JTabbedPane` is a Swing component that allows different components to be displayed on the same window by clicking on a tab. The tab appears at the top of the window. The `StudentWindow` from the previous sections is a subclass of `JFrame`. In all the examples except the first one, a `JPanel` called `mainPanel` is used to arrange all the GUI components to be displayed on the `StudentWindow`. We can easily modify `StudentWindow` so that it *becomes* the `mainPanel`. Four things must be done to modify `StudentWindow` and use a `JTabbedPane`:

(1) `StudentWindow` must inherit from `JPanel` rather than `JFrame`. Since it is now a `JPanel`, it is renamed `StudentPanel`.

(2) Instead of adding components to the `mainPanel` using its `add()` method (e.g., `mainPanel.add()`), the components are simply added to the `StudentPanel` using its `add()` method (e.g., `add()`), since the `StudentPanel` object now performs the role of `mainPanel`.

(3) A new class must be written to create the `JFrame` and attach the `JTabbedPane` to the `JFrame`. The class must also create the `StudentPanel` and add it to the `JTabbedPane`. This class is called `StudentApplication`, just as in the programming projects.

(4) The *Exit* button is removed from `StudentPanel` since it is better for the main window, `StudentApplication`, to manage the closing of the application.

To illustrate how the `JTabbedPane` works, it is necessary to create another `JPanel` called `CoursePanel`, which enables course information to be entered and edited. `CoursePanel` inherits from `JPanel` and uses a subset of the GUI components in `StudentPanel`. `StudentPanel` and `CoursePanel` use pull-from-above to access the domain objects. Figure 17.15 is a UML diagram of the main classes involved in generating the enhanced GUI. Figure 17.16 shows the classes in the domain layer.

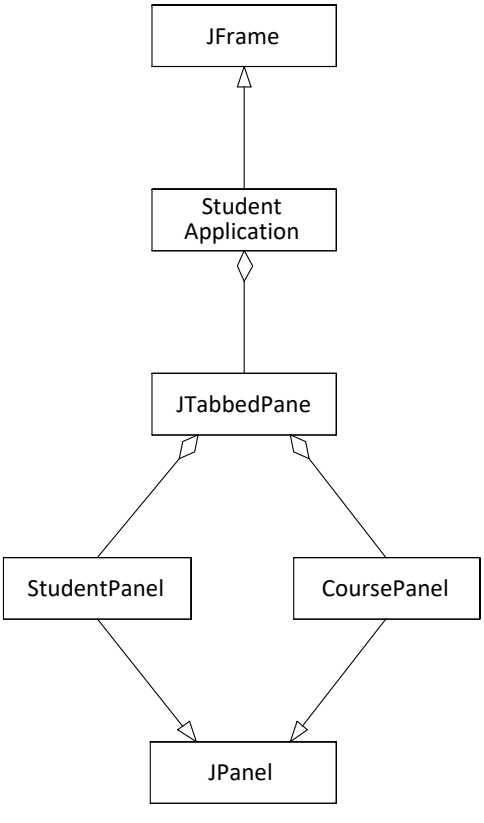

Figure 17.15: Main Classes in Enhanced GUI

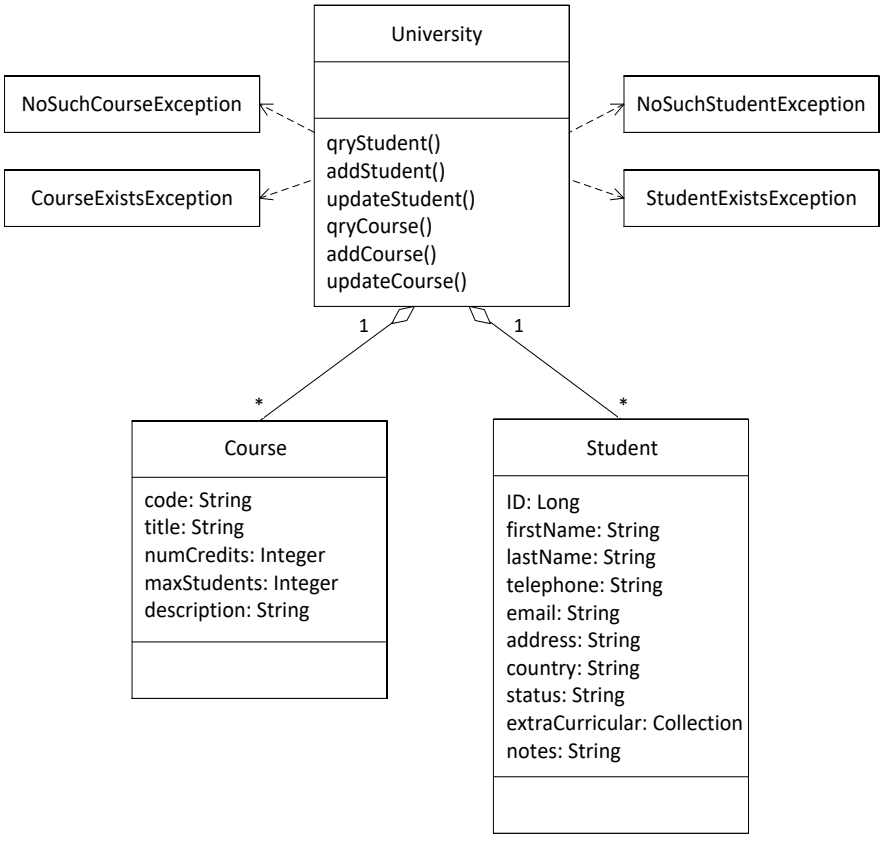

Figure 17.16: Domain Classes in Enhanced Student Application

The UML diagram of Figure 17.15 shows that `StudentApplication` is the main window and it inherits from `JFrame`. `StudentApplication` contains a `JTabbedPane` which contains two `JPanel`s, `StudentPanel` and `CoursePanel`. Both `StudentPanel` and `CoursePanel` are subclasses of `JPanel`.

The `StudentPanel` needs access to the `University` instance in order to manipulate `Student` objects. The `CoursePanel` also needs access to the `University` instance in order to manipulate `Course` objects. Thus, the `StudentApplication` also creates the single `University` instance and passes this instance as an argument to the `StudentPanel` and the `CoursePanel`.

The status bar in the main window displays informational messages generated from the main window and from `StudentPanel` and `CoursePanel`. Thus, `StudentPanel` and `CoursePanel` must be able to access the status bar in the

main window. The `StudentApplication` provides a method, `setStatus()` which can be used to set the status bar. A reference to the `StudentApplication` is passed to both `StudentPanel` and `CoursePanel` so that they can invoke the `setStatus()` method (and perhaps, make modifications to the GUI components in the main window displayed by `StudentApplication`).

17.8.2 Including Menus on the Main Window

The Swing component for a menu bar is `JMenuBar`. A menu bar can contain zero or more menus and each menu can contain zero or more menu items. The Swing component for a menu and menu item are `JMenu` and `JMenuItem`, respectively. Figure 17.17 shows a menu bar consisting of two menus, one of which contains three menu items.

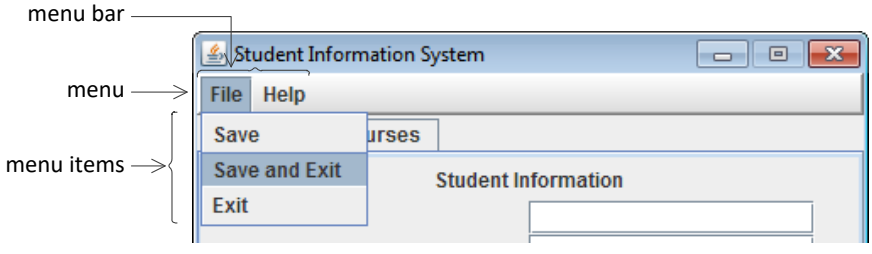

Figure 17.17 Menu Bar with Two Menus

To create the menu bar, the following code is used:

```
JMenuBar menuBar = new JMenuBar();
```

To create a menu called "File", the following code is used:

```
JMenu fileMenu = new JMenu("File");
```

Menu items are then created and attached to the menu:

```
JMenuItem saveMI = new JMenuItem("Save");
JMenuItem saveExitMI = new JMenuItem("Save and Exit");
JMenuItem exitMI = new JMenuItem("Exit");

fileMenu.add(saveMI);
fileMenu.add(saveExitMI);
fileMenu.add(exitMI);
```

Finally, the menu is attached to the menu bar:

```
menuBar.add(fileMenu);
```

When all the menus have been created and attached to the menu bar, the menu bar is attached to the `JFrame` using the following statement:

```
setJMenuBar(menuBar);
```

Note that it is not possible to attach a menu bar to a `JPanel`.

Figure 17.18 is a UML diagram showing the objects that are involved in producing the menu bar of the enhanced GUI.

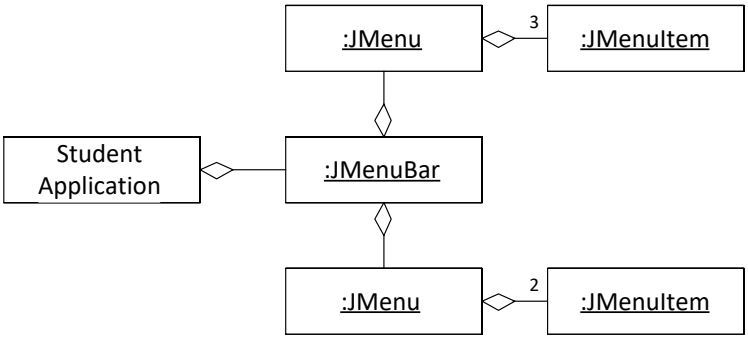

Figure 17.18: Objects that Produce Menu Bar in Student Application

Writing an event handler for a menu item is identical to writing one for a command button. Thus, in order to respond to the user clicking on a particular menu item, the `ActionListener` interface must be implemented. The `actionPerformed()` method can perform event handling just like with a command button. The `getActionCommand()` method of the `ActionEvent` parameter returns the text label of the menu item that was selected. Appropriate action can then be taken. If the menu item has the same text label as a command button, the same event handling code can work for both of them.

17.8.3 Popup Windows

Consider the popup window shown in Figure 17.19. This window can be generated using the class method `showMessageDialog()` from the

`JOptionPane` class. A `popup()` method can be used to generate a popup window whenever one is required. It is written as follows:

```
private void popup (String title, String message) {
   JOptionPane.showMessageDialog
      (null, message, title, JOptionPane.INFORMATION_MESSAGE);
}
```

There are several overloaded versions of the `showMessageDialog()` method; the Java API can be consulted for more information on each one.

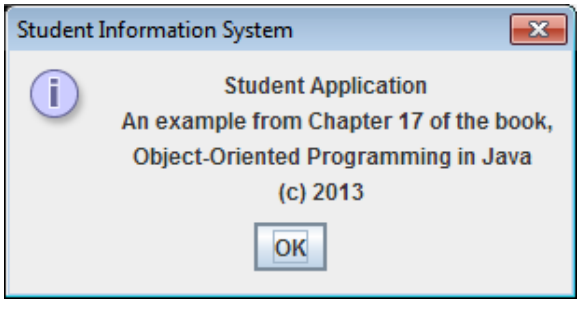

Figure 17.19: Popup Window

17.8.4 Completed Student Application

Figure 17.20 shows two screens from the final student application. Two menus are visible on the menu bar at the top. Two tabs are also visible at the top. Clicking on the "*Students*" tab causes the `StudentPanel` to be displayed. Clicking on the "*Courses*" tab causes the `CoursePanel` to be displayed. Most of the menu options cause a popup window to be displayed. To exit the application, the user can select the menu item "*Exit*" from the *File* menu. Alternatively, the user can click on the *Close* button.

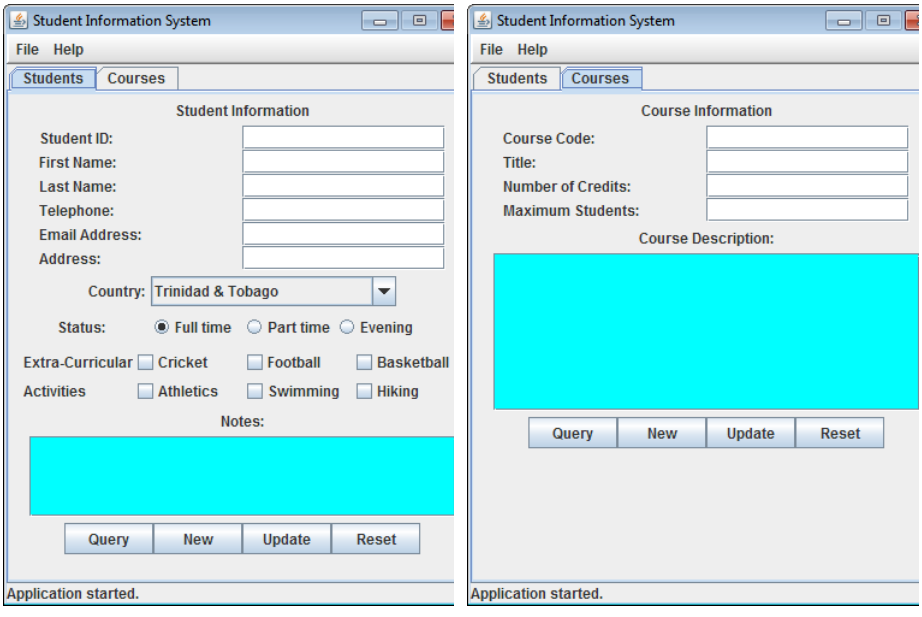

Figure 17.20: Windows of Completed Student Application

17.9 Is There a Better Way to Develop a GUI?

This chapter has described how to develop a GUI from scratch. The process can quickly become very tedious and error-prone when there are several GUI components and/or windows. Typically, a GUI developer would not follow the steps described in this chapter. Rather, a GUI developer will use a tool which simplifies the process of designing windows and populating them with the GUI components that are available. One such tool is *NetBeans GUI Builder*. This tool has features which make it easy to select GUI components from a palette and position them on a window. The tool also generates some of the code required for using the GUI components in a Java application. Of course, the event handling code is application-specific so it will have to be written by the developers of an application.

The objective of this chapter was not to describe an efficient way of developing a GUI for a Java application. Rather, the chapter sought to explain the object-oriented nature of GUI components and to show how they can be integrated with domain objects and objects in the persistence tier to achieve the functionality of an application. A good understanding of this process is important in order to design and build the software for the different layers of an object-oriented application.

Exercises

1. A GUI consists of one main window and several other child windows that are launched from the main window. The child windows occupy too much space on the monitor and it is convenient to display only the main window and one other window at any point in time. Explain how this can be achieved as efficiently as possible.

2. A GUI contains a menu bar with several menus. One of the menu items is "*Save*". It also has a button labeled "*Save*" which provides the same functionality as the menu item. Explain how the event handler for the *"Save*" feature should be implemented.

3. Should the domain objects in an object-oriented application contain user interface code? If not, explain how domain objects are created and manipulated via the user interface.

4. Suppose you wanted to design your own table for a GUI where the header and at most five rows of data should be displayed. The table should have three columns and at most 20 rows of data. It should be scrollable to permit the user to view the entire table. Using only the GUI objects discussed in this chapter, explain how you would implement the table.

5. Suppose you wanted the table component in (4) above to be available to other applications. Explain how you would go about creating the table as a reusable component so that a client can create the component by using only the `new` operator with the constructor, supplying arguments such as a data model (similar to a `JComboBox`) and a list of headings.

6. A certain GUI implements the `MouseListener` interface to trap mouse events on a window. However, whenever the mouse is clicked on the window, nothing happens. Explain what could be the reason for this.

7. Certain applications require two keys to be pressed in order to trigger the appropriate response from the domain objects. For example, in a game, pressing the right arrow with the up arrow indicates that the player would like to move in a northeasterly manner. However, the `keyPressed()` event handler is called *each* time a key is pressed on the keyboard (i.e., it is called once for each key press). Explain how the key combination can be detected by the event handler.

8. The `getGraphics()` method of the `JFrame` class returns a `Graphics` object which you can use to draw things on the window. The `Graphics` class has method `drawLine()` which takes four arguments representing the beginning and ending coordinates of the line and draws the line between the two points:

```
void drawLine(int x1, int y1, int x2, int y2)
//(x1, y1) and (x2, y2) are the end-points of the line
```

Extend the `SimpleWindow` class so that it displays a line from (0, 0) to the coordinates where the mouse is first clicked. After that, whenever the mouse is clicked, a line should be drawn from the previous mouse position to the current mouse position. Note that the current (`x`, `y`) coordinates of the mouse as well as the `Graphics` object should be instance variables of the `SimpleWindow` class.

9. Suppose that you wanted to design a special text field of your own. The `Graphics` class has a `drawRect()` method which takes four arguments to draw a rectangle: the top left hand coordinates of the rectangle and the `width` and `height` of the rectangle:

```
void drawRect (int x, int y, int width, int height)
//(x, y) is the upper left-hand coordinates of the rectangle
```

The `Graphics` class also has a `drawString()` method which draws a string at a certain (`x`, `y`) location on a window:

```
drawString(String s, int x, int y);
// draws s at the coordinates (x, y)
```

Use these two methods to design your own text field (as simple as possible). You will need to implement the `KeyListener` interface. Characters should be displayed on the text field as they are typed on the keyboard. The backspace key should be used to delete characters from the right.

10. The top-level windows in this chapter inherit from `JFrame`. Another approach is to let the top-level windows *contain* a `JFrame` (i.e., use composition rather than inheritance). Evaluate the strengths and weaknesses of both approaches.