Chapter 10

Graphical User Interfaces and Event-Driven Programming

Just as the connection points between program components are called interfaces, the "connection point" between a program and its human user is called its "user interface." A program that uses visual aids—buttons, scroll bars, menus, etc.—to help a user enter input and read output is called a graphical user interface *("GUI" for short, pronounced "goo-ee").*

In this chapter, we learn the following:

- *how to employ Java's AWT/Swing framework to design graphical user interfaces*
- *how to write programs whose controllers are* distributed *and* event driven. *That is, a program has multiple controllers, and a controller executes when it is started by a user action (*event*), e.g., a button press.*
- *how to use the* observer *design pattern to streamline the collaborations between the components of a program that uses a GUI.*

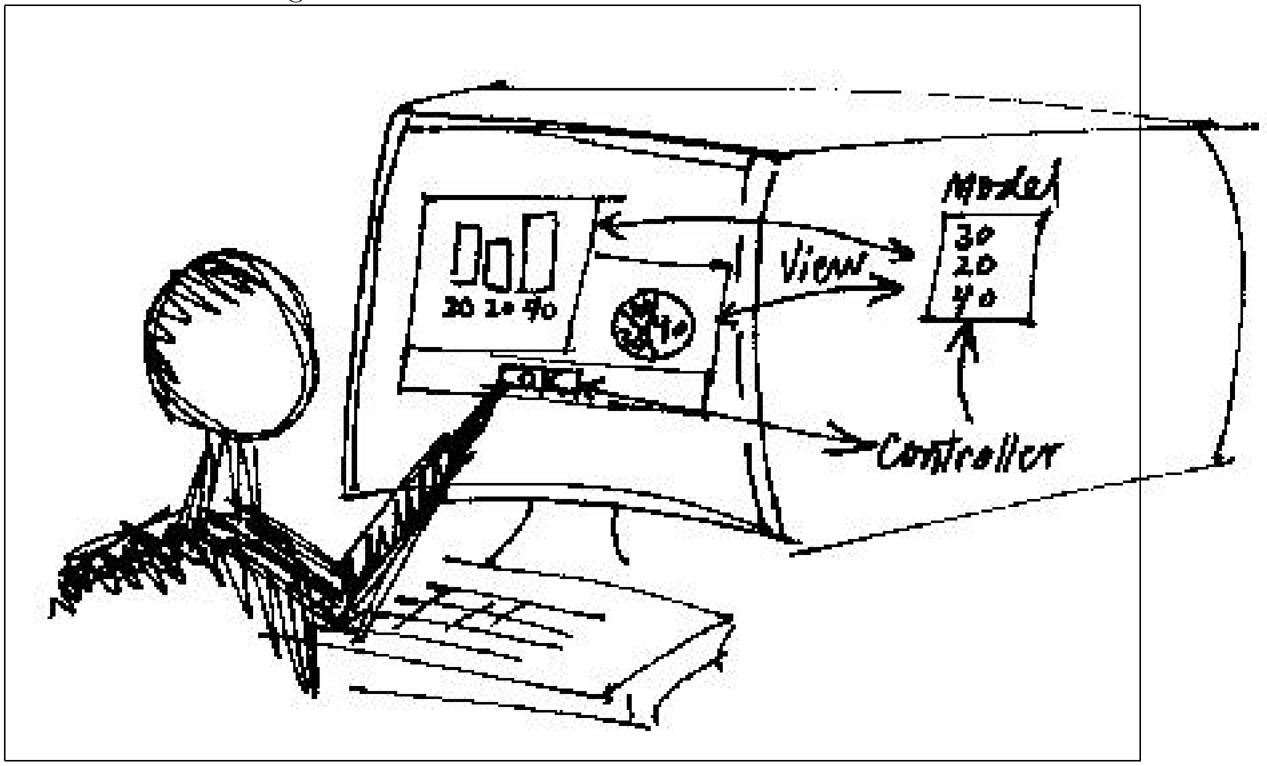
10.1 Model-View-Controller Revisited

The model-view-controller (MVC) architecture we have used for our programs was first developed to manage programs with GUIs. The philosophy and terminology behind MVC might be explained as follows:

- A computer program is a kind of "appliance," like a television set, radio, or hand-held calculator, so it should look like one: The program should have a *view* or appearance like an appliance, so that its human user feels familiar with it.
- Appliances have controls—switches, buttons, knobs, sliders—that their users adjust to operate it. A program should also have *controllers* that its user adjusts to execute the program. And for familiarity's sake, the controllers should have the appearance of switches, buttons, knobs, etc.
- An apppliance's controls are connected to the appliance's internal circuitry, which does the work that its user intended. Similarly, a program's controllers are connected to the program's *model*, which calculates results that are portrayed by the program's view. By "portrayed," we mean that *the view presents a picture of the internal state of the model.* Indeed, a program's model might even have multiple views that are presented simultaneously.

The manner in which the model, view, and controller collaborate is equally important: A user interacts with a program's view, say, by adjusting one of its controllers. This awakens the controller, which might examine the view for additional data and then send messages to the program's model. The model executes the controller's messages, computing results and updating its internal state. The view is then sent a

Figure 10.1: model-view-controller architecture



message (either indirectly by the model or directly by the controller) to display the results. The view queries the model, getting information about the model's state and presenting this information to the user. Figure 1 depicts the situation.

Another important aspect about the MVC-architecture is that it can be *composed*, that is, one component—say, the view—might be built from smaller components—say, buttons, text fields, and menus—that are themselves constructed with their own little MVC-architectures. For example, consider a text field, which is a component within a view where a user can type text. The text field has its own little appearance or view. (Typically, it is an area that displays the letters the user typed while the mouse was positioned over the area.) When a user types a letter into the text field, this activates the text field's controller, which transfers the letter into the text field's internal model. Then, the text field's view is told to refresh itself by asking the model for all the letters that have been typed; the view displays the letters.

Of course, it would be a nightmare if we must design from scratch the little MVC architectures for buttons, menus, text fields, and the other GUI components! For this reason, we use a framework that contains prebuilt components and provides interfaces for connecting them. In Java, the framework for building GUIs and connecting them to controllers and models is called *AWT/Swing*. (The AWT part is the `java.awt` package; the Swing part is the `javax.swing` package. "AWT" stands for

"Abstract Window Toolkit.") The bulk of this chapter introduces a useful subset of AWT/Swing.

Exercise

Consider a television set. What are its controllers? model? view? Answer the same questions for a calculator. How can a television have multiple views? How can a calculator have multiple views?

10.2 Events

In the programs we built in previous chapters, the program's controller was "in control"—it controlled the sequence of steps the user took to enter input, it controlled the computation that followed, and it controlled the production of the program's output. When one employs a GUI, this changes—the program's user decides when to enter input, and the controllers must react accordingly. The moving of the mouse, the pushing of a button, the typing of text, and the selection of a menu item are all forms of input data, and the controllers must be prepared to calculate output from these forms of input. The new forms of input are called *events*, and the style of programming used to process events is called *event-driven programming.*

Event-driven programming is more complex than the programming style we employed in earlier chapters: When a program receives events as input, a coherent "unit" of input might consist of multiple events (e.g., a mouse movement to a menu, a menu selection, text entry, mouse movement to a button, and a button push); the program that receives this sequence of events must be written so that it can

- process each individual event correctly—this is called *handling the event.* The controller that handles the event is sometimes called the *event handler* or *event listener.*
- accumulate information from handling the sequence of events and generate output. Typically, the controllers that handle the events save information about them in model objects.

Further, the user of the program might generate events in unexpected or incorrect orders, and event handlers must be prepared to handle unwelcome events.

The previous examples should also make clear that event-driven programs use multiple controllers (event handlers), so there is no longer one controller that oversees the execution of the entire program. Instead, execution is distributed across multiple controllers that are activated at the whim of a user. For this reason, an event-driven program is a bit like a crew of night-duty telephone operators who are repeatedly awakened by telephone calls (events) and must process each call in a way that keeps the telephone station operating smoothly through the night. The telephone operators cannot predict when the telephone will ring and what each call's request might be; the operators must react to the evening's events rather than dictate what they might be.

To assist an event driven program, a programmer must design the program's GUI so that

- sequences of events are organized into natural units for processing
- it is difficult or impossible for the user to generate a truly nonsensical sequence of events

To help with the first objective, we will usually design our GUIs so that computation occurs only after a sequence of events terminates with a button push or a menu selection. (In Java, button pushes and menu selections are called *action events*, and their event handlers are called *action listeners*.) We will let mouse-movement and text-entry events be handled by the default event handlers that are already built into the components of the Java AWT/Swing framework.

To assist with the second objective, we will design our GUIs so that the action events that can be generated from a window can occur in any order at all. If a program must enforce that one action event must occur before another, we will program the GUI so that the first action event causes a secondary window to appear from which the second action event can be generated.

Because we let the AWT/Swing components handle mouse movement events and text entry events, we need not write extra controllers to monitor the position of the mouse on the display or to monitor every key press into a text field—we use the code that is already in place in the default codings of windows and text fields. This should make clear why a framework is so useful for building graphical user interfaces: Many intelligent classes are available for immediate use, and we need worry only about programming controllers for those events (here, action events) that we choose to handle in a customized way.

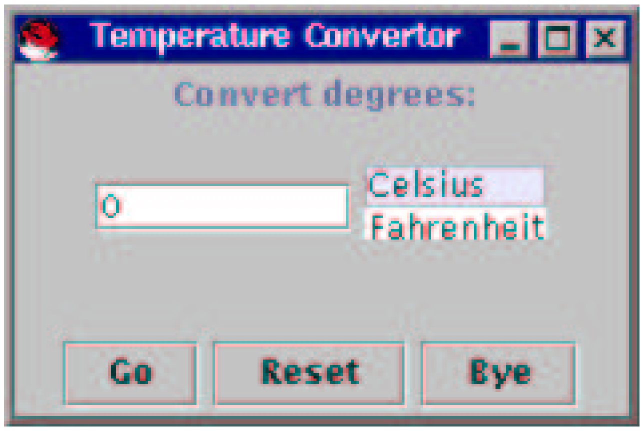
10.3 The AWT/Swing Class Hierarchy

Before we write GUIs with Java's AWT/Swing framework, we must survey the components provided by the framework. We begin with some terminology.

An entity that can have a position and size (on the display screen) and can have events occur within it is called a *component*. A component that can hold other components is a *container*; a *panel* is the standard container into which one inserts components (including painted text and shapes). Panels are themselves inserted into a container called a *window*, which is a "top-level" container, that is, a container that can be displayed by itself.

A *frame* is a window with a title and menus; frames are "permanent" in that they are created when an application starts and are meant to exist as long as the application

Figure 10.2: frame with basic components



executes. A *dialog* is a "temporary" window that can appear and disappear while the program is executing.

Examples of components that one finds within panels and frames are

- a *label*, which is text that the user can read but cannot alter
- a *text component*, into which a user can type text
- a *button*, which can be pushed, triggering an action event
- a *list* of items, whose items can be chosen ("selected")

Figure 2 shows a frame that contains a label, a text component, a list, and three buttons. Although it is not readily apparent, the text component and list are embedded in a panel, which was inserted in the middle of the frame, and the label and button live in their own panels in the top and bottom regions of the frame.

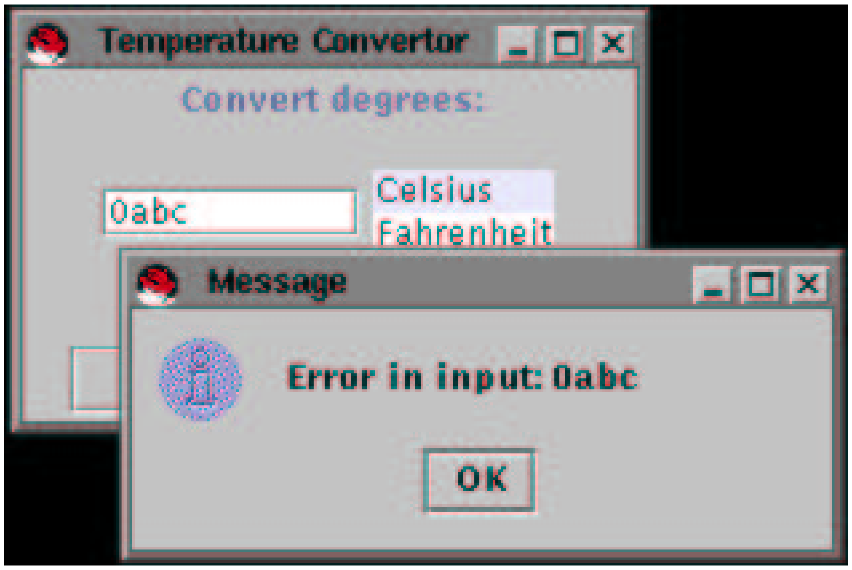
Figure 3 displays an example dialog, which might have appeared because the user entered text and pushed a button in the frame behind. The dialog contains a label and a button; when the button is pushed, the dialog disappears from the screen.

A frame can hold a *menu bar*, which holds one or more *menus*. Each menu contains *menu items*, which can be selected, triggering an action event. Figure 4 shows a frame with a menu bar that holds two menus, where the second menu is open and displays four menu items, one of which is itself a menu:

When a container holds multiple components, the components can be formatted with the assistance of a *layout manager*. Three basic forms of layout are

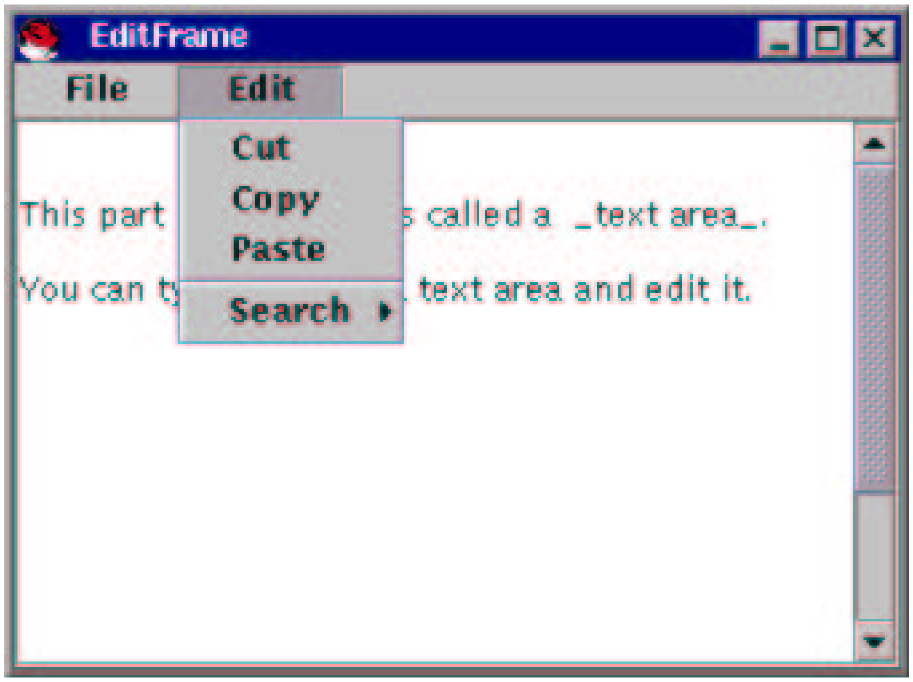
- *flow layout*: the components are arranged in a linear order, like the words in a line of text

Figure 10.3: sample dialog



Error in input: 0abc

Figure 10.4: frame with menus



This part s called a _text area_.

You can t text area and edit it.

- *border layout*: components are explicitly assigned to the "north," "south," "east," "west," or "center" regions of the container
- *grid layout*: components are arranged as equally-sized items in rows and columns, like the entries of a matrix or grid

In Figure 2, the frame is organized with a 3-by-1 grid layout, where the second element of the grid is a panel containing a text component and list. The panel uses flow layout to arrange its two components. We will see an example of border layout momentarily.

The AWT/Swing framework contains dozens of classes, each of which owns dozens of methods. It is overwhelming to learn the entire framework, so we will master a manageably sized subset of it. Figure 5 displays the parts of AWT/Swing we will use; aside from reading the names of the various classes, *do not study the Figure at this point*—use it as a reference as you progress through the chapter.

Since the AWT/Framework was developed in several stages by the Java designers, the elegant hierarchy of component-container-panel-window-frame is obscured in the final product in Figure 5.

The classes in Figure 5 require other classes that define points, type fonts, images, layouts, and events. Figure 6 lists these extra classes. Again, study the Figure as you progress through the chapter.

As all frameworks must do, AWT/Swing uses a variety of interfaces for connecting view components to controllers. (Review Chapter 9 for uses of Java `interfaces`; an introductory explanation appears later in this chapter, also.) The interfaces within AWT/Swing that we employ appear in Figure 7.

Figures 5 through 7 list the constructs we use in this chapter, and you should use the Figures as a road map through the examples that follow.

Exercises

1. List all the methods owned by a `JFrame`. (Remember that the `JFrame` inherits the methods of its superclasses.) Compare your answer to Table 21, Chapter 4.
2. Several of the classes in Figure 5 have methods named `addSOMETHINGListener`. The classes that possess such methods are capable of generating `SOMETHING` events. List the classes that generate events.

10.4 Simple Windows: Labels and Buttons

The standard graphical user interface for a program is a frame, generated from `class JFrame`. We made extensive use of `JFrame` in previous chapters, using it to hold and display panels. Now, we learn how to insert components like labels and buttons into a frame.

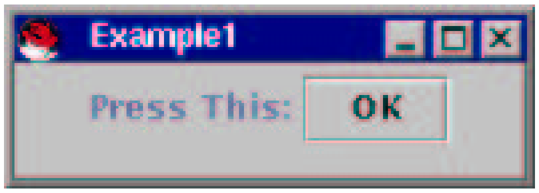
Figure 10.5: partial AWT/Swing hierarchy

```
Object
 |
 +-Component [abstract]:  setSize(int,int), setVisible(boolean), setFont(Font),
    |     isShowing():boolean, getLocationOnScreen():Point, setLocation(Point),
    |     paint(Graphics), repaint(), setForeground(Color),
    |     setBackground(Color), getGraphics()
    |
    +-Container: add(Component), add(Component,Object), setLayout(LayoutManager)
       |
       +-Window: pack(), dispose(), addWindowListener(WindowListener)
       |  |
       |  +-JFrame: JFrame(), setTitle(String), setJMenuBar(JMenuBar)
       |            getContentPane():Container
       |
       +-JApplet: JApplet(), init(), getParameter(String):String,
       |          getContentPane():Container, setJMenuBar(JMenuBar)
       |
       +-JComponent [abstract]: paintComponent(Graphics)
          |
          +-AbstractButton [abstract]: addActionListener(ActionListener),
          |  |   setEnabled(Boolean), getText():String, setText(String),
          |  |   setSelected(Boolean), isEnabled():boolean,
          |  |   isSelected():boolean, doClick()
          |  |
          |  +-JButton: JButton(String), JButton(Icon), JButton(String, Icon),
          |  |   setIcon(Icon)
          |  |
          |  +-JMenuItem: JMenuItem(String)
          |     |
          |     +-JMenu: JMenu(String), add(Component), addSeparator()
          |
          +-JLabel: JLabel(String), getText():String, setText(String)
          |
          +-JList: JList(Object[]),getSelectedIndex():int,setSelectedIndex(int),
          |  getSelectedIndices():int[], setSelectedIndices(int[]),
          |  setSelectionMode(int), clearSelection(),
          |  addListSelectionListener(ListSelectionListener)
          |
          +-JMenuBar: JMenuBar(), add(JMenu)
          |
          +-JOptionPane: showMessageDialog(Component,Object),
          |              showConfirmDialog(Component,Object):int,
          |              showInputDialog(Component,Object):String
        ...
```

Figure 10.5: partial AWT/Swing hierarchy (concl.)

```
            |
            +-JPanel: Panel(), Panel(LayoutManager)
            |
            +-JScrollPane: JScrollPane(Component)
            |
            +-JTextComponent [abstract]: cut(), paste(), copy(), getText():String,
               |   setText(String), getSelectionStart():int,
               |   getSelectionEnd():int, getSelectedText():String,
               |   replaceSelection(String), getCaretPosition():int,
               |   setCaretPosition(int), moveCaretPosition(int),
               |   isEditable():boolean, setEditable(boolean)
               |
               +-JTextField: JTextField(String,int),
               |             addActionListener(ActionListener)
               +-JTextArea: JTextArea(String,int,int), insert(String,int)
                            replaceRange(String,int,int), setLineWrap(boolean)
```

The first example is a frame that holds the label, `Press This`, and a button named `OK`. These two components must be inserted into the frame, and we must indicate the form of layout. We use flow layout, which places the two components next to each other. Here is the result,



which is produced by the program in Figure 8.

Let's examine the statements in the constructor method one by one:

- `JLabel label = new JLabel("Press This:")` constructs a label object, `label` that displays the string, `Press This:`.
- Similarly, `JButton button = new JButton("OK")` constructs a button.
- `Container c = getContentPane()` asks the frame to extract (the address of) its *content pane* and assign it to `c`. Many examples in earlier chapters invoked `getContentPane`, and now it is time to understand the activity.

  For the moment, pretend that a `JFrame` object is in fact a real, physical, glass window that is assembled from several layers of glass. The frame's topmost

Figure 10.6: points, fonts, images, layouts, and events

```
Object
 |
 +-BorderLayout: BorderLayout(), BorderLayout.NORTH, BorderLayout.SOUTH,
 |               BorderLayout.EAST, BorderLayout.WEST, BorderLayout.CENTER
 |
 +-FlowLayout: FlowLayout(), FlowLayout(int), FlowLayout.LEFT,
 |             FlowLayout.RIGHT, FlowLayout.CENTER
 |
 +-GridLayout: GridLayout(int,int)
 |
 +-Font: Font(String,int,int), Font.PLAIN, Font.BOLD, Font.ITALIC
 |
 +-Point: Point(int,int), translate(int,int)
 |
 +-EventObject
 |  |
 |  +-AWTEvent [abstract]
 |  |  |
 |  |  +-ActionEvent: getActionCommand():String, getSource():Object
 |  |  |
 |  |  +-WindowEvent
 |  |
 |  +-ListSelectionEvent
 |
 +-WindowAdapter [implements WindowListener]
 |
 +-Image
 |
 +-ImageIcon [implements Icon]: ImageIcon(String), getImage():Image
 |
 +-Observable: addObserver(Observer), setChanged(), notifyObservers(),
               notifyObservers(Object)
```

Figure 10.7: interfaces

```
public interface ActionListener
{ public void actionPerformed(ActionEvent e); }

public interface WindowListener
{ public void windowActivated(WindowEvent e);
  public void windowClosed(WindowEvent e);
  public void windowClosing(WindowEvent e);
  public void windowDeactivated(WindowEvent e);
  public void windowDeiconified(WindowEvent e);
  public void windowIconified(WindowEvent e);
  public void windowOpened(WindowEvent e);
}

public interface ListSelectionListener
{ public void valueChanged(ListSelectionEvent e); }

public interface Observer
{ public void update(Observable ob, Object arg); }

public interface Icon
{ public int getIconHeight();
  public int getIconWidth();
  public void paintIcon(Component c, Graphics g, int x, int y);
}
```

Figure 10.8: frame with label and button

```
import java.awt.*;
import javax.swing.*;
/** Frame1 is a frame with a label and a button */
public class Frame1 extends JFrame
{ /** Constructor  Frame1  creates a frame with a label and button */
  public Frame1()
  { JLabel label = new JLabel("Press This:");
    JButton button = new JButton("OK");
    Container c = getContentPane();
    c.setLayout(new FlowLayout());
    c.add(label);
    c.add(button);
    setTitle("Example1");
    setSize(200, 60);
    setVisible(true);
  }

  public static void main(String[] args)
  { new Frame1(); }
}
```

layer is called the *glass pane*, and it is possible (but not recommended) to paint on it. When we insert components like panels, labels, and buttons into the frame, the components are inserted into an inner layer, called the *content pane*. (There are additional layers of "glass," but we will not deal with them in this chapter.)

The statement, `Container c = getContentPane()`, fetches the content pane. We could also write the statement as

```
Container c = this.getContentPane();
```

but from this point onwards, we take advantage of Java's convention: a message that an object sends to itself need not be prefixed by `this`.

- The message, `c.setLayout(new FlowLayout())` tells the content pane to arrange the components in a flow layout, that is, where the components are arranged in a line.

- `c.add(label)` uses the content pane's `add` method to add `label`; `c.add(button)` adds `button` also.

- Finally, the `setTitle`, `setSize`, and `setVisible` messages are the standard ones. Again, we omit the `this` pronoun as the receiver.

The names, `label` and `button`, are not crucial to the example, and we might revise the above statements to read,

```
Container c = getContentPane();
c.setLayout(new FlowLayout());
c.add(new JLabel("Press This:"));
c.add(new JButton("OK"));
```

Indeed, even the name, `c`, can be discarded, e.g., `getContentPane().setLayout(new FlowLayout())`, `getContentPane().add(new JLabel("Press This:"))`, etc.

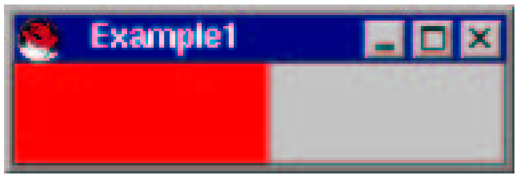
The statement, `setSize(200, 60)`, which sets the frame's size, can be replaced by `pack()`, which resizes the frame at a minimal size to display the components it contains. (But take care when using `pack()`, because it occasionally creates a frame with too small of a size for its components.)

As is the custom, we place a tiny `main` method at the end of the class so that we can easily test the frame.

As noted earlier, it is possible to paint on the surface of a frame by using a method named `paint`. if we add this method

```
public void paint(Graphics g)
{ g.setColor(Color.red);
  g.fillRect(0, 0, 100, 100);
}
```

to Figure 8, we get this result,



because we have painted on the frame's topmost, "glass," pane, covering the components we inserted into the content pane.

Although we should not paint directly onto a frame, we can set the background and foreground colors of the content pane and the components we insert into it. For example, to make the content pane's background yellow, we state,

```
Container c = getContentPane();
c.setBackground(Color.yellow);
```

and we can make `button` a matching yellow by saying,

```
button.setBackground(Color.yellow);
```

We can color red the foreground text on both `label` and `button` by saying

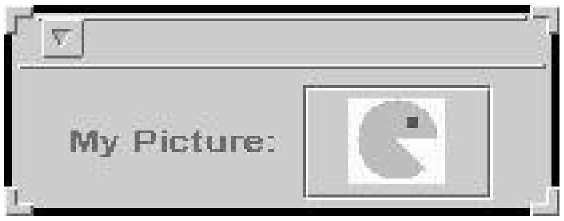
```
label.setForeground(Color.red);
button.setForeground(Color.red);
```

This works because every component has a "background" and "foreground" that can be colored.

Buttons usually have text displayed on their faces, but it is possible to display images, called *icons*, as well. If you have an image formatted as a `gif` or `jpg` file, you can place the image on the face of the button as follows:

```
ImageIcon i = new ImageIcon("mypicture.gif");
JButton b = new JButton(i);
```

which displays



That is, the image file, `mypicture.gif`, is used to construct an `ImageIcon` object, which itself is used to construct the button. A button can display both an image and text:

```
JButton b = new JButton("My Picture:", new ImageIcon("mypicture.gif"));
```

Once the frame and its button appear on the display, you can move the mouse over the frame's button and push it to your heart's content. But the program takes no action when the button is pushed, because the program has only a view object but no model and no controller. We write these next.

Exercises

1. Create a frame with three buttons whose labels state, `Zero`, `One`, and `Two`. Create the frame with different sizes, e.g., `setSize(300, 150)`, `setSize(50, 300)`, and `pack()`.

2. Color the frame's background white, color each button's background a different color, and color each button's text (foreground) black.

3. Replace the three buttons with one label that states, `Zero One Two`. Color the background and foreground of the label.

10.5 Handling an Event

Every component—button, label, panel, etc.—of a window is an object in its own right. When an event like a button push or a mouse movement occurs, it occurs within an object within the window. The object in which the event occurs is the *event source*. In AWT/Swing, when an event occurs, the event source automatically sends a message to an object, called its *event listener*, to handle the event. The event listener is a controller—when it receives a message, it sends messages to model object and view object to compute and display results.

As stated earlier, we focus upon *action events*—button pushes and menu selections. An action event, like a button push, is handled by an *action-listener* object; in Java, an action-listener object must have a method named `actionPerformed`, and it is this method that is invoked when an action event occurs.

When we write a class, `C`, that creates action-listener objects, we use this format:

```
public class C implements ActionListener
{ ...

  public void actionPerformed(ActionEvent e)
  { ... instructions that handle a button-push event ... }
}
```

That is, the class's header line asserts `implements ActionListener`, and the class contains an `actionPerformed` method. As we learned in Chapter 9, we use a Java `interface` to name the methods required of a class. In AWT/Swing, there is a prewritten interface, named `ActionListener`, which is found in the `java.awt.event` package. The interface was listed in Figure 7; once again, it looks like this:

```
/** ActionListener names the method needed by an action listener. */
public interface ActionListener
{ /** actionPerformed handles an action event, say, a button push
    * @param e - information about the event */
  public void actionPerformed(ActionEvent e);
}
```

The interface states that an action listener must have a method named `actionPerformed`. AWT/Swing requires that a button's action listener *must* be an object that is constructed from a class that `implements ActionListener`. We now study three standard ways of writing action listeners.

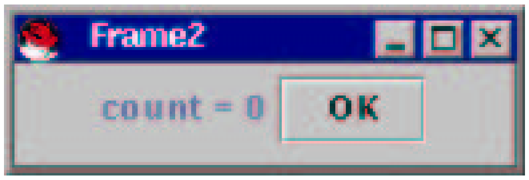
10.5.1 A View as Action Listener

For our first, simplest example of event handling, we alter `Frame1` so that each time the the frame's button is pushed, its label displays the total number of button pushes.

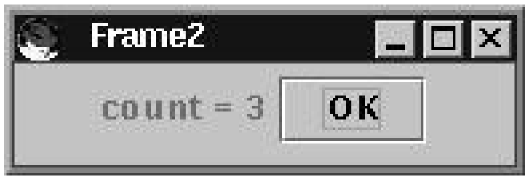
Figure 10.9: model class, Counter

```
/** Counter  holds a counter */
class Counter
{ private int count;   // the count

  /** Constructor  Counter  initializes the counter
    * @param start - the starting value for the count  */
  public Counter(int start)
  { count = start; }

  /** increment  updates  count. */
  public void increment()
  { count = count + 1; }

  /** countOf accesses count.
    * @return the value of  count */
  public int countOf()
  { return count; }
}
```

When created, the GUI appears



and after 3 button pushes, the view is



To create this behavior, we use a model-view-controller architecture, where `class Counter`, from Figure 9, remembers the quantity of button pushes; it acts as the model.

Next, we require a controller (action listener) for the `OK` button, and we must connect the controller to the button. In this first example, the view and controller are combined into the same class; this is a naive and inelegant but simple solution—see Figure 10.

Figure 10.10: combined view/controller for counter example

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/** Frame2a shows a frame with whose label displays the number of times
    its button is pushed */
class Frame2a extends JFrame implements ActionListener
{ private Counter count;  // address of model object
  private JLabel label = new JLabel("count = 0");  // label for the frame

  /** Constructor  Frame2a creates a frame with a label and button
    * @param c - the model object, a counter */
  public Frame2a(Counter c)
  { count = c;
    Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(label);
      JButton button = new JButton("OK");
      cp.add(button);
      button.addActionListener(this); // this  object---the view---is connected
                                      // to  button  as its action listener
    setTitle("Frame2a");
    setSize(200, 60);
    setVisible(true);
  }

  /** actionPerformed handles an action event---a button push */
  public void actionPerformed(ActionEvent e)
  { count.increment();
    label.setText("count = " + count.countOf());
  }
}


/** Example2a starts the application */
public class Example2a
{ public static void main(String[] args)
  { Counter model = new Counter(0);     // create the model
    Frame2a view = new Frame2a(model);  // create the controller and view
  }
}
```

Since the view is also the controller, `class Frame2a` states in its header line that it `implements ActionListener`. And indeed, method `actionPerformed` appears at the end of the class. (We study the contents of `actionPerformed` momentarily.) Also, the statement, `import java.awt.event.*`, appears at the beginning of the class because the `java.awt.event` package contains the `ActionListener` interface.

Now, how do we connect the button to its action listener, the view object? The answer appears within the view's constructor method:

```
JButton button = new JButton("OK");
cp.add(button);
button.addActionListener(this);
```

The first two statements create the button and add it to the view, like before. The third statement connects `button` to its action listener—every button has a method, named `addActionListener`, which is invoked to connect a button to its action listener. Here, the action-listener object is `this` very object—the view object, which displays the button! From this point onwards, every push of `button` causes the view object's `actionPerformed` method to execute.

`Frame2a`'s `actionPerformed` method handles a button push by making the counter increment and by resetting the text of the label to display the counter's new value. The method receives a parameter that contains technical information about the button push; we will not use the parameter at this time.

Here is a slightly detailed explanation of what happens when the program in Figure 10 is started and its button is pushed. When `Example2a` is started:

1. The `main` method creates objects for the model and the view/controller. `Frame2a`'s constructor method creates a label and a button. The button is sent an `addActionListener` message that tells the button that its action events will be handled by the `Frame2a` object.

2. The view appears on the display, and the program awaits events.

When the user clicks on the frame's `OK` button:

1. The computer's operating system detects the event, and tells AWT/Swing about it. AWT/Swing determines the event source (the button), and creates an `ActionEvent` object that holds precise information about the event. The `ActionEvent` object is sent as the actual parameter of an `actionPerformed` message to the event source's action listener, which is the `Frame2a` object.

2. The message arrives at `actionPerformed` in `Frame2a`. The `actionPerformed` method sends an `increment` message to the counter object and a `setText` message to the label.

3. When `actionPerformed`'s execution concludes, *the computer automatically refreshes the view on the display*; this displays the label's new value.

4. The program awaits new events.

The explanation should make clear that *computation occurs when events trigger execution of event listeners.* This is why computation is "event driven."

10.5.2 A Separate Controller

Our second approach to the the previous example uses a separate class to be the button's event listener—its controller. This style is preferred because it makes it easier to build GUIs with multiple buttons and to reuse views and controllers. The controller, `class CountController`, will be studied momentarily. This leaves the view, `class Frame2b`, with the sole job of presenting the frame, label, and button on the display. Figure 11 shows the view after we have extracted the controller from it.

The key change in the constructor method lies at

```
JButton button = new JButton("OK");
button.addActionListener(new CountController(count, this));
```

which constructs a new `CountController` object (see Figure 12), gives it the addresses of the model object and view object (`this` object), and attaches the controller to the button as the button's action listener.

Because the controller is separate from the view, the former will need a way to tell the latter when it is time to display a new value for the count. For doing this, the view class has a method, `update`, which resets the text of the label with the latest value of the count.

We have simple coding of the controller, `class CountController`, in Figure 12.

The controller `implements ActionListener` and it holds the `actionPerformed` method that is invoked when the `OK` button is pushed. This controller resembles the ones we saw in previous chapters, because its job is to tell the model object to compute results and tell the view to display the results.

10.5.3 A Button-Controller

The third solution to the example merges the `JButton` object with its controller. This follows the philosophy that, to the user, the button *is* the controller, and pushing the button activates its methods. The button-controller we write appears in Figure 13.

`CountButton` is a "customized" `JButton`, hence it `extends JButton`. Its constructor method starts work by invoking the constructor in `JButton`—this is what `super(my_label)` does—and creates the underlying `JButton` and attaches `my_label` to the button's face.

But the constructor for `CountButton` does more: The crucial statement is, `addActionListener(this)`, which tells the button that its action listener is `this` very same object. Thus, the class `implements ActionListener`.

Figure 10.11: view class for counter example

```
import java.awt.*;  import javax.swing.*;
/** Frame2b shows a frame with whose label displays the number of times
    its button is pushed */
public class Frame2b extends JFrame
{ private Counter count;  // address of model object
  private JLabel label = new JLabel("count = 0");  // label for the frame

  /** Constructor  Frame2b creates a frame with a label and button
    * @param c - the model object, a counter */
  public Frame2b(Counter c)
  { count = c;
    Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(label);
      JButton button = new JButton("OK");
      button.addActionListener(new CountController(count, this)); // see Fig. 12
      cp.add(button);
    setTitle("Frame2");
    setSize(200, 60);
    setVisible(true);
  }

  /** update revises the view */
  public void update()
  { label.setText("count = " + count.countOf()); }
}

/** Example2b starts the application */
public class Example2b
{ public static void main(String[] args)
  { Counter model = new Counter(0);
    Frame2b view = new Frame2b(model);
  }
}
```

Figure 10.12: controller for counter example

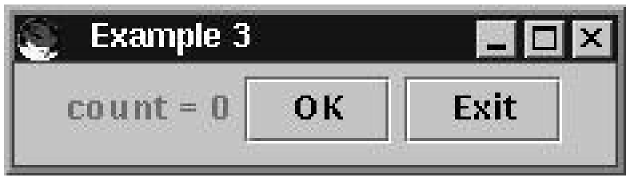
```
import java.awt.event.*;
/** CountController handles button push events that increment a counter */
public class CountController implements ActionListener
{ private Frame2b view;  // the view that must be refreshed
  private Counter model;  // the counter model

  /** CountController constructs the controller
    * @param my_model - the model object
    * @param my_view - the view object  */
  public CountController(Counter my_model, Frame2b my_view)
  { view = my_view;
    model = my_model;
  }

  /** actionPerformed handles a button-push event */
  public void actionPerformed(ActionEvent evt)
  { model.increment();
    view.update();
  }
}
```

The view that uses the button-controller is in Figure 14, and it is the simplest of the three versions we have studied.

Figure 15 summarizes the architecture that we have assembled for the third variant of the counter application.

The interface, written in italics, serves as the connection point between the button-controller and the AWT/Swing framework.

We can revise the application in Figure 15 so that it it presents two buttons, an `OK` button that changes the count, and an `Exit` button that terminates the application:



The result holds interest because there are now *two* controllers, one for each action event. The controller for terminating the program appears in Figure 16. (Recall that `System.exit(0)` terminates an application.)

Next, Figure 17 defines the view, `class Frame3`, as a quick extension of `class Frame2c` from Figure 14. Although an abstract class, like those in Chapter 9, might be a better solution for organizing the previous and revised views, we use the existing

Figure 10.13: button-controller for counter example

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/** CountButton defines a button-controller */
public class CountButton extends JButton implements ActionListener
{ private Frame2c view;   // the view that holds this controller
  private Counter model;  // the model that this controller collaborates with

  /** Constructor CountButton builds the controller
    * @param my_label - the label on the button that represents the controller
    * @param my_model - the model that the controller collaborates with
    * @param my_view - the view that the controller updates  */
  public CountButton(String my_label, Counter my_model, Frame2c my_view)
  { super(my_label);  // attach label to the button in the superclass
    view = my_view;
    model = my_model;
    addActionListener(this); // attach this very object as the ``listener''
  }

  /** actionPerformed handles a push of this button
    * @param evt - the event that occurred, namely, the button push */
  public void actionPerformed(ActionEvent evt)
  { model.increment();
    view.update();
  }
}
```

Figure 10.14: view for Example2

```
import java.awt.*;
import javax.swing.*;
/** Frame2c shows a frame with whose label displays the number of times
    its button is pushed */
public class Frame2c extends JFrame
{ private Counter count;  // address of model object
  private JLabel label = new JLabel("count = 0");  // label for the frame

  /** Constructor  Frame2c creates a frame with a label and button
    * @param c - the model object, a counter */
  public Frame2c(Counter c)
  { count = c;
    Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(label);
      cp.add(new CountButton("OK", count, this)); // the button-controller
    setTitle("Example2");
    setSize(200, 60);
    setVisible(true);
  }

  /** update revises the view */
  public void update()
  { label.setText("count = " + count.countOf()); }
}


/** Example2c starts the application */
public class Example2c
{ public static void main(String[] args)
  { Counter model = new Counter(0);
    Frame2c view = new Frame2c(model);
  }
}
```

Figure 10.15: architecture of Example2c

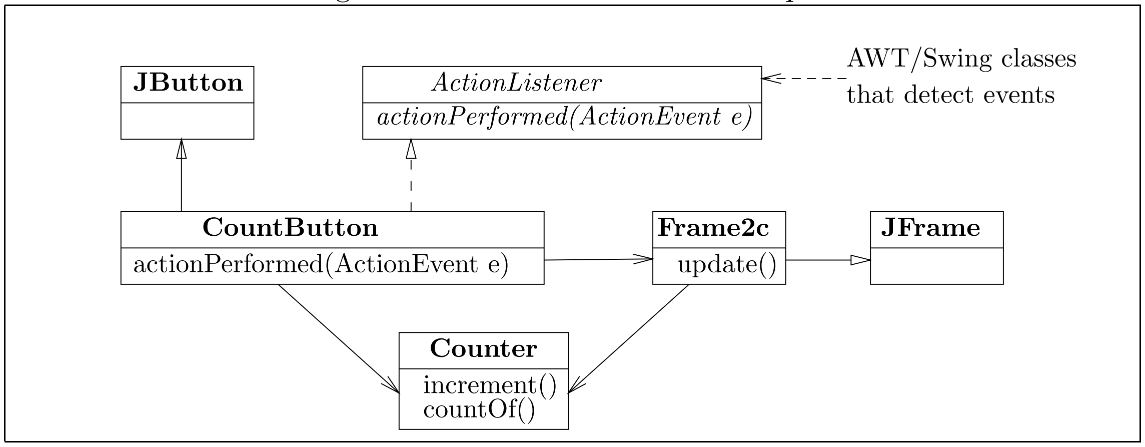


Figure 10.16: exit controller

```
import javax.swing.*;
import java.awt.event.*;
/** ExitButton defines a controller that terminates an application */
public class ExitButton extends JButton implements ActionListener
{
  /** Constructor ExitButton builds the controller
    * @param my_label - the label for the controller's button  */
  public ExitButton(String my_label)
  { super(my_label);
    addActionListener(this);
  }

  /** actionPerformed  handles a button-push event
    * @param evt - the event  */
  public void actionPerformed(ActionEvent evt)
  { System.exit(0); }
}
```

Figure 10.17: view for two-button view

```
import java.awt.*;
import javax.swing.*;
/** Frame3 shows a frame with whose label displays the number of times
    its button is pushed */
class Frame3 extends Frame2c
{
  /** Constructor  Frame3 creates a frame with a label and button
    * @param c - the model object, a counter */
  public Frame3(Counter c)
  { super(c);  // tell superclass to construct most of the frame
    Container cp = getContentPane();
    cp.add(new ExitButton("Exit"));  // add another button-controller
    setTitle("Example 3");  // reset the correct title and size:
    setSize(250, 60);
    setVisible(true);
  }
}


/** Example3 starts the application */
public class Example3
{ public static void main(String[] args)
  { Counter model = new Counter(0);
    Frame3 view = new Frame3(model);
  }
}
```

approach if only to show that components can be added to a frame's content pane incrementally. (See `Frame3`'s constructor method.)

Finally, users of Windows-style operating systems have the habit of terminating applications by clicking the "X" button that appears at a window's upper right corner. The buttons at the upper right corner are monitored through the AWT/Swing `WindowListener` interface. (See Figure 7 for the interface.) Since it does not deserve a long explanation, we merely note that the changes presented in Figure 18 will make a mouse click on the "X" button terminate a program.

Exercises

1. Create an interface for an application whose model possesses two `Counter` objects. Create two buttons, each of which increments one of the counters when pushed. Use labels to display the values of the two counters.

Figure 10.18: terminating a program with the X-button

```
// Define this class:
import java.awt.event.*;
public class ExitController extends WindowAdapter
{ public void windowClosing(WindowEvent e)
  { System.exit(0); }
}


// Within the frame's constructor method, add this statement:
addWindowListener(new ExitController());
```

2. As noted in Figure 5, buttons have a `setText(String s)` method, which changes the text that appears on a button's face to `s`. Revise your solution to the previous exercise so that each button displays the number of times that it has been pushed.

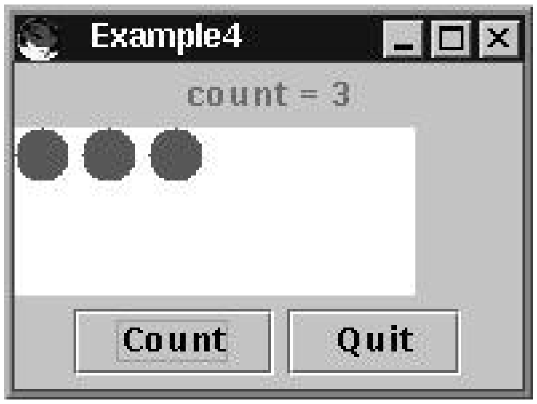
3. Create an application whose GUI has an `Increment` button, a `Decrement` button, and label that displays an integer. Each time `Increment` is pushed, the integer increases, and each time `Decrement` is pushed, the integer decreases. (Hint: Write a new model class.)

4. Create a GUI with a red button, a yellow button, and a blue button. The frame's background turns into the color of the button last pushed.

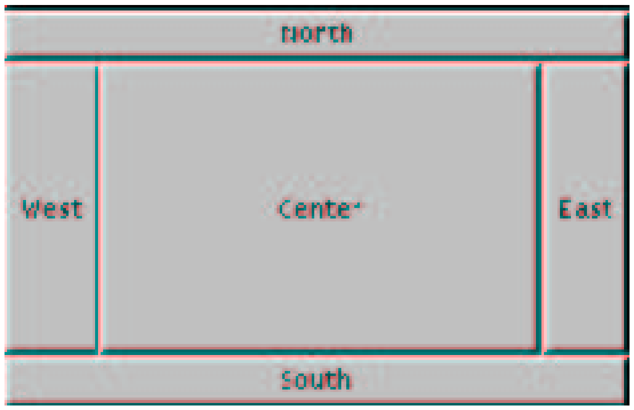
10.6 Richer Layout: Panels and Borders

We can rebuild the above GUI so that its label appears at the top of the frame, the `OK` and `Exit` buttons are positioned along the frame's bottom, and a drawing of the current count appears in the center. Figure 19 shows this. To do this, we use *panels*. We construct a panel, insert the label in it, and place the panel in the "north" region of the frame. Next, we construct another panel, paint it, and place it in the frame's "center" region. Finally, we insert the two buttons into a third panel and place the panel in the "south" region. *Border layout* lets us specify the regions where components can be inserted; in addition to the region used in the example in Figure 19, there are also "east" and "west" regions. Here is a picture of how the regions are

Figure 10.19: frame with two depictions of the count



positioned in a container by the border layout manager:



The north and south regions take precedence in layout, followed by the east and west regions. Any space left in the middle becomes the center region. The layout is delicate in the sense that an undersized window may cause components in a region to be partially hidden.

Figure 20 presents the view for the GUI in Figure 19.

In `Frame4`'s constructor, the content pane is told to use border layout:

```
Container cp = getContentPane();
cp.setLayout(new BorderLayout());
```

Next, components are added to the regions of the content pane, e.g.,

```
cp.add(drawing, BorderLayout.CENTER);
```

In addition to the regions used in the Figure, one can also use `BorderLayout.EAST` and `BorderLayout.WEST`.

Panels are simply created and inserted into the frame, e.g.,

Figure 10.20: view class with border layout and panels

```
import java.awt.*;
import javax.swing.*;
/** Frame4 is a frame with a label and a button */
public class Frame4 extends JFrame
{ private Counter count;  // address of model object
  private JLabel lab = new JLabel("count = 0");  // label for the frame
  private JPanel drawing; // a drawing for the center of the frame

  /** Constructor  Frame4 creates a frame with label, drawing, and 2 buttons
    * @param c - the model object, a counter
    * @param panel - a panel that displays a drawing  */
  public Frame4(Counter c, JPanel panel)
  { count = c;
    drawing = panel;
    Container cp = getContentPane();
      cp.setLayout(new BorderLayout());
      JPanel p1 = new JPanel(new FlowLayout());
        p1.add(lab);
      cp.add(p1, BorderLayout.NORTH);
      cp.add(drawing, BorderLayout.CENTER);
      JPanel p2 = new JPanel(new FlowLayout());
        p2.add(new CountButton("Count", count, this));
        p2.add(new ExitButton("Quit"));
      cp.add(p2, BorderLayout.SOUTH);
    setTitle("Example4");
    setSize(200,150);
    setVisible(true);
  }

  /** update revises the view */
  public void update()
  { lab.setText("count = " + count.countOf());
    drawing.repaint();
  }
}
```

Figure 10.20: view class with border layout and panels (concl.)

```
import java.awt.*;
import javax.swing.*;
/** Drawing creates a panel that displays a small drawing */
public class Drawing extends JPanel
{ private Counter count;  // the model object

  public Drawing(Counter model)
  { count = model;
    setSize(200, 80);
  }

  public void paintComponent(Graphics g)
  { g.setColor(Color.white);
    g.fillRect(0, 0, 150, 80);
    g.setColor(Color.red);
    for ( int i = 0; i != count.countOf(); i = i+1 )
        { g.fillOval(i * 25, 0, 20, 20); }
  }
}

/** Example4 starts the application */
public class Example4
{ public static void main(String[] args)
  { Counter model = new Counter(0);
    Drawing drawing = new Drawing(model);
    Frame4 view = new Frame4(model, drawing);
  }
}
```

```
JPanel p2 = new JPanel(new FlowLayout());
  p2.add(new CountButton("Count", count, this));
  p2.add(new ExitButton("Quit"));
cp.add(p2, BorderLayout.SOUTH);
```

The first statement simultaneously creates a panel and sets its layout. Components are added to panels just like they are to frames, and panels are added to frames just like other components are added.

Class `Drawing` extends `JPanel`. It repaints the drawing when told.

An image file can be displayed within a panel by using the panel's graphics pen. Insert these statements into the panel's `paintComponent` method:

```
ImageIcon i = new ImageIcon("mypicture.gif");
Image j = i.getImage();
```

```
g.drawImage(j, 20, 20, this);
```

The first statement converts the image file into an ImageIcon, as we did for creating labels for buttons. The second statement extracts an Image object from the icon, and the third employs the panel's graphics pen to draw the image with its upper left corner at position 20, 20.

Exercises

1. Test Example4 by pushing its Count button 10 times. What happens to the drawing? Resize the frame to a larger size; what do you observe? Propose a solution to this problem.

2. Experiment with border layout: Rewrite Frame4 in Figure 20 so that the three panels are inserted into the east, center, and west regions, respectively; into the center, north, and east regions, respectively. Next, delete panel p1 from the constructor method and insert the label directly into the north region.

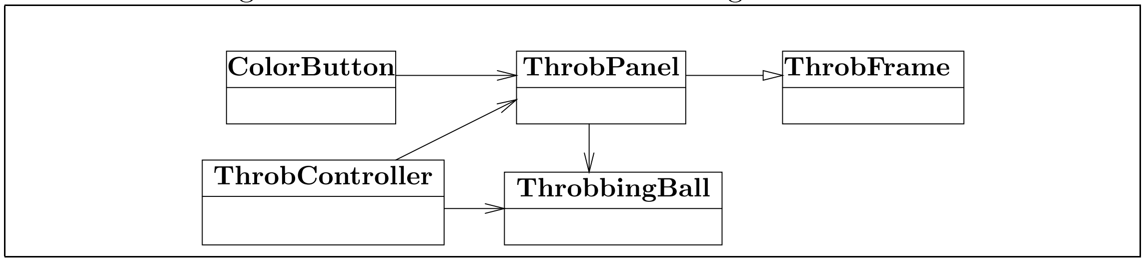
3. Write a application that lets you grow and shrink an egg: The GUI has two buttons, Grow and Shrink. An egg is displayed underneath the buttons; when you push Grow, the egg gets 10% larger; when you push Shrink, the egg becomes 10% smaller. Use method paintAnEgg from Figure 2, Chapter 5, to paint the egg.

10.6.1 An Animation in a Panel

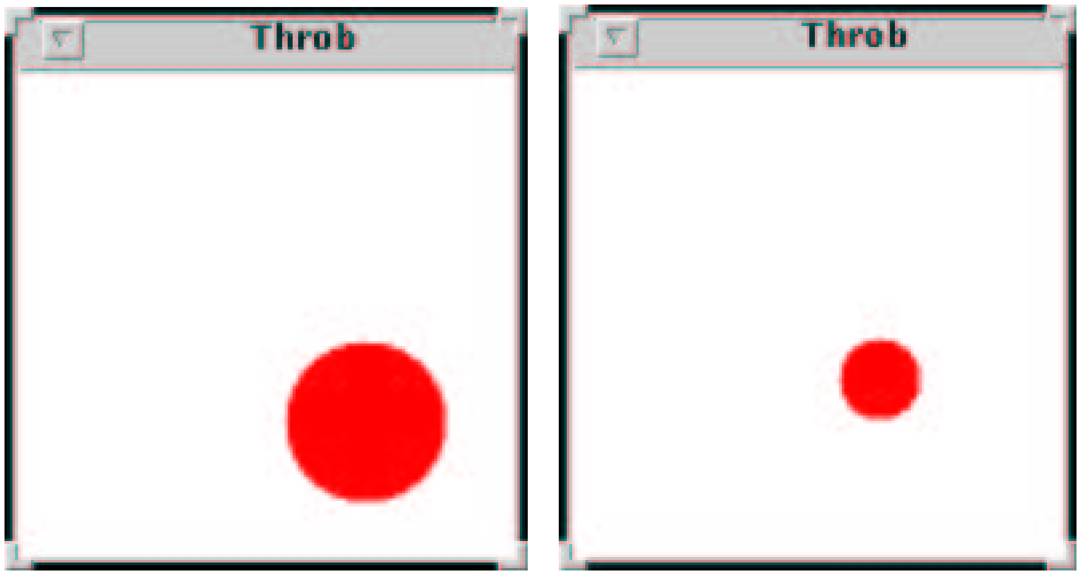
Animations, like the moving ball example in Chapter 7, can be easily reformatted to appear within a panel of a GUI. The GUI might also display buttons that, when pressed, alter the animation's progress—video games are built this way.

Here is a simple example. Perhaps we have an animation that displays a ball that

Figure 10.21: architecture of throbbing ball animation



"throbs" between large and small:



When the button is pressed, the ball in the animation changes color; in this simple way, the user "plays" the animation.

The animation is displayed in a panel that is embedded into a frame. Like the animation in Chapter 7, this animation has its own controller, and there is an additional button-controller for changing the color of the ball. Figure 21 displays the architecture. The ball is modelled by `class ThrobbingBall`; the view consists of `ThrobPanel`, which paints the ball on a panel, and `ThrobFrame`, which displays the panel and the color-change button on a frame. Figure 22 displays these classes.

The application's controllers hold the most interest: `ThrobController` contains a `run` method whose loop resizes the throbbing ball and redraws it on the panel; `ColorButton` changes the ball's color each time its button is pressed. The controllers are connected to the model and view by means of the start-up class, `StartThrob`. All three classes appear in Figure 23.

Because the `ThrobController`'s `run` method is a nonterminating loop, it is crucial that this method is invoked as the last statement in the animation. (The section,

Figure 10.22: model and view classes for animation

```
/** ThrobbingBall models a ball that changes size from large to small */
public class ThrobbingBall
{ private boolean is_it_currently_large;  // the ball's state---large or small

  public ThrobbingBall() { is_it_currently_large = true; }

  /** isLarge returns the current state of the ball */
  public boolean isLarge() { return is_it_currently_large; }

  /** throb makes the ball change state between large and small  */
  public void throb() { is_it_currently_large = !is_it_currently_large; }
}

import java.awt.*;
import javax.swing.*;
/** ThrobPanel draws a throbbing ball */
public class ThrobPanel extends JPanel
{ private int panel_size;       // size of this panel
  private int location;         // where ball will be painted on the panel
  private int ball_size;        // the size of a ``large'' ball
  private Color c = Color.red;  // the ball's color
  private ThrobbingBall ball;   // the ball object

  public ThrobPanel(int size, ThrobbingBall b)
  { panel_size = size;
    location = panel_size / 2;
    ball_size = panel_size / 3;
    ball = b;
    setSize(panel_size, panel_size);
  }

  /** getColor returns the current color of the ball */
  public Color getColor() { return c; }

  /** setColor resets the color of the ball to  new_color */
  public void setColor(Color new_color) { c = new_color; }

 ...
```

Figure 10.22: model and view classes for animation (concl.)

```
  /** paintComponent paints the ball */
  public void paintComponent(Graphics g)
  { g.setColor(Color.white);
    g.fillRect(0, 0, panel_size, panel_size);
    g.setColor(c);
    if ( ball.isLarge() )
         { g.fillOval(location, location, ball_size, ball_size); }
    else { g.fillOval(location, location, ball_size / 2, ball_size / 2); }
  }
}


import java.awt.*;
import javax.swing.*;
/** ThrobFrame displays the throbbing-ball panel and color-change button */
public class ThrobFrame extends JFrame
{ /** Constructor  ThrobFrame  builds the frame
    * @param size - the frame's width
    * @param p - the panel that displays the ball
    * @param b - the color-change button */
  public ThrobFrame(int size, ThrobPanel p, ColorButton b)
  { Container cp = getContentPane();
    cp.setLayout(new BorderLayout());
    cp.add(p, BorderLayout.CENTER);
    cp.add(b, BorderLayout.SOUTH);
    setTitle("Throb");
    setSize(size, size + 40);
    setVisible(true);
  }
}
```

Figure 10.23: Controllers for animation

```
/** ThrobController runs the throbbing-ball animation */
public class ThrobController
{ private ThrobPanel writer; // the output-view panel
  private ThrobbingBall ball;  // the ball model object
  private int time;  // how long animation is delayed before redrawn

  /** ThrobController initializes the controller
    * @param w - the panel that is controlled
    * @param b - the ball that is controlled
    * @param delay_time - the amount of time between redrawing the animation */
  public ThrobController(ThrobPanel w, ThrobbingBall b, int delay_time)
  { writer = w;
    ball = b;
    time = delay_time;
  }

  /** run  runs the animation  forever */
  public void run()
  { while ( true )
          { ball.throb();
            writer.repaint();  // redisplay ball
            delay();
          }
  }

  /** delay pauses execution for  time  milliseconds */
  private void delay()
  { try { Thread.sleep(time); }
    catch (InterruptedException e) { }
  }
}
```

Figure 10.23: Controllers for animation (concl.)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/** ColorButton controls the color of the ball */
public class ColorButton extends JButton implements ActionListener
{ private ThrobPanel view;  // the view object where shapes are drawn

  public ColorButton(ThrobPanel f)
  { super("OK");
    view = f;
    addActionListener(this);
  }

  /** actionPerformed handles a click */
  public void actionPerformed(ActionEvent e)
  { Color c = view.getColor();
    if ( c == Color.red )
         { view.setColor(Color.blue); }
    else { view.setColor(Color.red); }
  }
}

/** StartThrob assembles the objects of the animation */
public class StartThrob
{ public static void main(String[] a)
  { int frame_size = 180;   // size of displayed frame
    int pause_time = 200;   // speed of animation (smaller is faster)
    ThrobbingBall b = new ThrobbingBall();
    ThrobPanel p = new ThrobPanel(frame_size, b);
    ThrobFrame f = new ThrobFrame(frame_size, p, new ColorButton(p));
    new ThrobController(p, b, pause_time).run();   // important: do this last!
  }
}
```

"Threads of Execution," at the end of this chapter explains why.) It is also crucial that the loop within `run` has a `delay`, because it is during the period of delay that the computer detects action events and executes action listeners.

Exercises

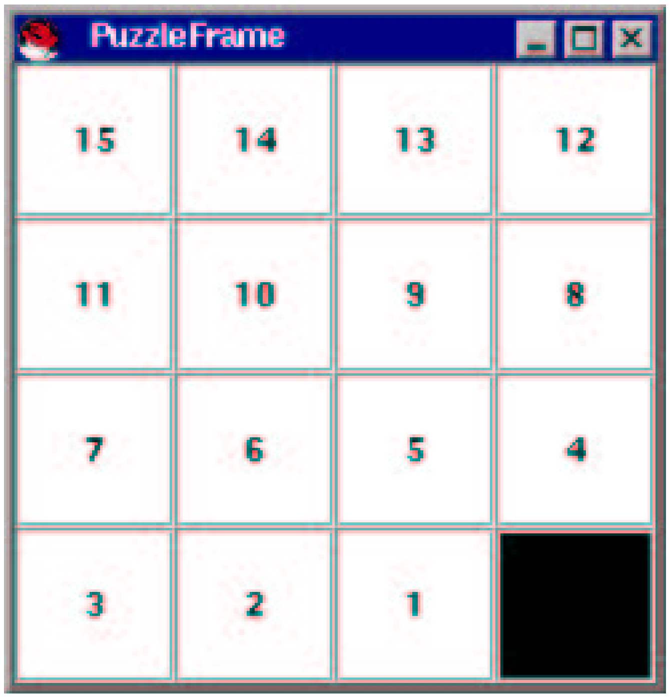
1. Add a "Pause" button to the throbbing-ball animation that, when pushed, causes the ball to stop throbbing until the button is pushed again. (Hint: add another boolean field variable to `ThrobbingBall`.)

2. Swap the last two statements of `main` in `StartThrob`. Explain what the animation no longer operates.

3. Embed the moving-ball animation of Figure 7, Chapter 7 into a panel; insert the panel into a frame. Next, add two buttons, "Faster" and "Slower," which increase and decrease, respectively, the velocity at which the ball travels in the animation. (You will have to write additional methods for `class MovingBall` in Figure 8, Chapter 7; the methods will adjust the ball's x- and y-velocities.)

10.7 Grid Layout

When you have a collection of equally-sized components to be arranged as a table or grid, use grid layout. As an example, consider the slide puzzle program from Figure 11, Chapter 8, where its output view, `class PuzzleWriter`, is replaced by a GUI where buttons portray the pieces of the puzzle—a click on a button/piece moves the puzzle piece. The view might look like this:



The frame is laid out as a 4-by-4 grid, where each cell of the grid holds a button-controller object created from `class PuzzleButton` (which will be studied momentarily). Figure 24 shows `class PuzzleFrame`, the puzzle's GUI.

A grid layout of `m` rows by `n` columns is created by `new GridLayout(m, n)`. Adding components to a grid layout is done with the `add` method, just like with flow layout. In the constructor method in the Figure, the nested for-loop creates multiple distinct `PuzzleButton` objects and inserts them into a grid layout; the grid is filled row by row. As explained in the next paragraph, it is helpful to retain the addresses of the button objects in an array, named `button`, but such an array is not itself required when using grid layout.

Each button displays a numerical label on its face, stating the numbered piece it represents. A user "moves" a slide piece by clicking on it. Unfortunately, the buttons themselves do *not* move—the labels on the buttons move, instead. When a button is clicked, its event handler sends a message to method `update`, which consults array `button` and uses the `setBackground` and `setText` methods to repaint the faces of the buttons.

Figure 25 displays the controller class, `PuzzleButton`. When pushed, the button's `actionPerformed` method uses `getText()` to ask itself the number attached to its face; it then tries to move that number in the model. If the move is successful, the view is told to update.

The controller in Figure 25 should be contrasted with the one in Figure 12, Chapter 8. The Chapter 8 controller used one central loop to read integers and make moves

Figure 10.24: grid layout for slide puzzle game

```
import java.awt.*;  import javax.swing.*;
/** PuzzleFrame shows a slide puzzle */
public class PuzzleFrame extends JFrame
{ private SlidePuzzleBoard board;   // the model; see Fig. 10, Ch. 8
  private int size;                 // the board's size
  private int button_size = 60;     // width/height of each button
  private PuzzleButton[][] button;  // the buttons on the face of the view

  /** Constructor PuzzleFrame builds the view
    * @param board_size - the width and depth of the puzzle
    * @param b - the model, a slide puzzle board  */
  public PuzzleFrame(int board_size, SlidePuzzleBoard b)
  { size = board_size;
    board = b;
    button = new PuzzleButton[size][size];
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(size, size));
    // create the button-controllers and insert them into the layout:
    for ( int i = 0; i != size; i = i+1 )
        { for ( int j = 0; j != size; j = j+1 )
              { button[i][j] = new PuzzleButton(board, this);
                cp.add(button[i][j]);
              }
        }
    update();  // initialize the pieces with their numbers
    addWindowListener(new ExitController()); // activates X-button; see Fig. 15
    setTitle("PuzzleFrame");
    setSize(size * button_size + 10,  size * button_size + 20);
    setVisible(true);
  }

  /** update  consults the model and repaints each button */
  public void update()
  { PuzzlePiece[][] r = board.contents();  // get contents of the puzzle
    for ( int i = 0; i != size; i = i+1 )  // redraw the faces of the buttons
        { for ( int j = 0; j != size; j = j+1 )
              { if ( r[i][j] != null )
                       { button[i][j].setBackground(Color.white);
                         button[i][j].setText("" + r[i][j].valueOf()); }
                else { button[i][j].setBackground(Color.black);
                         button[i][j].setText( "" );
                       }
              }
        }
  }
}
```

Figure 10.24: grid layout for slide puzzle game (concl.)

```
/** Puzzle creates and displays the slide puzzle */
public class Puzzle
{ public static void main(String[] args)
  { int size = 4;  // a  4 x 4  slide puzzle
    SlidePuzzleBoard board = new SlidePuzzleBoard(size); // see Fig. 10, Ch. 8
    PuzzleFrame frame = new PuzzleFrame(size, board);
  }
}
```

Figure 10.25: button-controller for slide puzzle

```
import javax.swing.*;
import java.awt.event.*;
/** PuzzleButton implements a button controller for a puzzle game */
public class PuzzleButton extends JButton implements ActionListener
{ private SlidePuzzleBoard puzzle; // address of the SlidePuzzle model
  private PuzzleFrame view;  // address of Frame that displays this button

  /** Constructor PuzzleButton builds the button
    * @param my_puzzle - the address of the puzzle model object
    * @param my_view - the address of the puzzle's view */
  public PuzzleButton(SlidePuzzleBoard my_puzzle, PuzzleFrame my_view)
  { super(""); // set label to nothing, but this will be repainted by the view
    puzzle = my_puzzle;
    view = my_view;
    addActionListener(this);
  }

  /** actionPerformed processes a move of the slide puzzle */
  public void actionPerformed(ActionEvent evt)
  { String s = getText();  // get the number on the face of this button
    if ( !s.equals("") )   // it's not the blank space, is it?
       { boolean ok = puzzle.move(new Integer(s).intValue()); // try to move
         if ( ok ) { view.update(); }
       }
  }
}
```

forever. But the just class just seen is used to construct multiple controllers, each of which is programmed to make just one move. *There is no loop to control the moves.* Instead, the application is controlled by the user, who can push buttons forever. Whenever the user tires, the application patiently waits for more events.

Exercises

1. Experiment with grid layout: Rewrite `Frame4` in Figure 20 so that the three components are added into a frame with `new GridLayout(3,1)`; with `new GridLayout(1,3)`; with `new GridLayout(2,2)`.

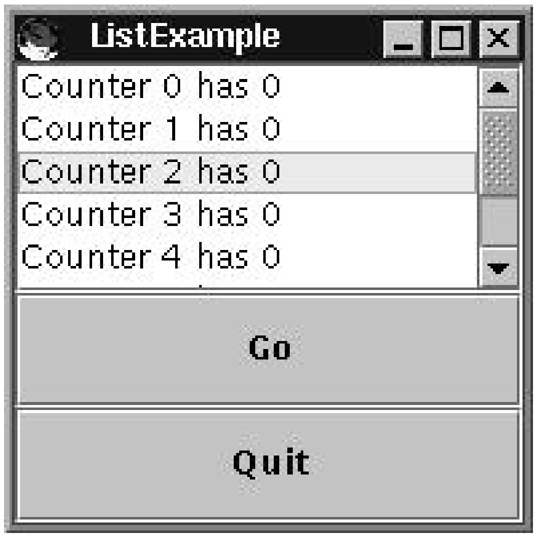
2. Add to the GUI in Figure 24 a label that displays the count of the number of pieces that have been moved since the slide-puzzle was started. (Hint: Use the counter from Figure 9.)

3. Create a GUI that looks like a calculator. The GUI displays 9 buttons, arranged in a 3-by-3 grid, and a numerical display (label) attached to the top.

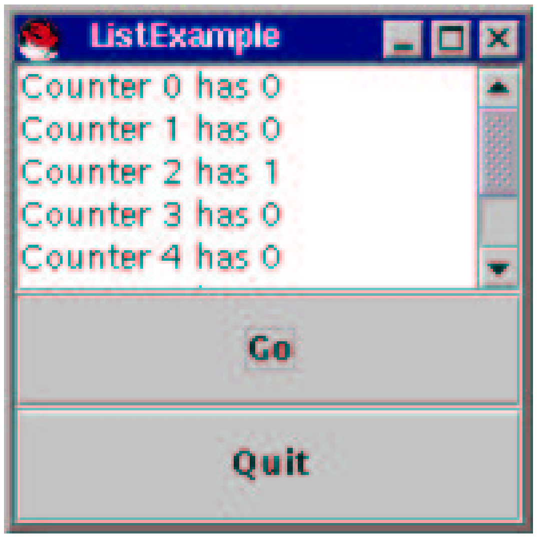
10.8 Scrolling Lists

A scrolling list (or "list", for short) can be used when a user interface must display many items of information. The scrolling list has the added advantage that a user can click on one or more of its items, thus highlighting or *selecting* them. (Selecting a list item generates an event—not an action event, but a *list selection event.* The default event listener for a list selection event highlights the selected item.) At a subsequent button push, the list's items can be examined and altered.

Here is a small example that uses a scrolling list. The list displays the values of eight counters. When a user clicks on a list item, it is highlighted. Here, the third item is selected:



When the user pushes the `Go` button, an action event is triggered, and the associated action listener increments the counter associated with the selected item. For example, if the `Go` button was clicked on the above interface, we obtain this new view:



In this manner, a scrolling list can present choices, like buttons do, and give useful output, like labels do.

A scrolling list is built in several steps: You must create a "model" of the list, you must insert the model into a `JList` object, and you must insert the `JList` into a `JScrollPane` object to get a scroll bar. These steps might look like this:

```
String[] list_labels = new String[how_many];  // the list's ``model''
JList items = new JList(list_labels);         // embed model into the list
JScrollPane sp = new JScrollPane(items);      // embed list into a scroll pane
```

Now, `sp` can be added into a content pane, e.g.,

```
Container cp = getContentPane();
 ...
cp.add(sp);
```

Whatever strings that are assigned to the elements of `list_labels` will be displayed as the items of the list, e.g.,

```
for ( int i = 0;  i != list_labels.length;  i = i+1 )
    { list_labels[i] = "Counter " + i + " has 0"; }
```

At the beginning of the chapter, we noted that each AWT/Swing component is built as a little MVC-architecture of its own. When we create a `JList` object, we must supply its model part, which must be an array of objects, e.g., strings like `list_labels` just seen. If the objects are not strings, then we must ensure that the objects have

Figure 10.26: model and view for scrolling list of counters

```
/** ListExample displays an array of counters as a scrolling list */
public class ListExample
{ public static void main(String[] a)
  { int how_many_counters = 8;
    Counter2[] counters = new Counter2[how_many_counters];  // the model
    for ( int i = 0;  i != how_many_counters;  i = i+1 )
        { counters[i] = new Counter2(0, i); }               // see below
    new ListFrame(counters);                                // the view
  }
}

/** Counter2 is a Counter that states its identity with a  toString  method */
public class Counter2 extends Counter
{ private int my_index;

  public Counter2(int start, int index)
  { super(start);
    my_index = index;
  }

  public String toString()
  { return "Counter " + my_index + " has " + countOf(); }
}
```

a `toString` method, which `JList`'s internal view uses to display the objects as list items.

Here is how we build the example program displayed above:

- We extend `class Counter` in Figure 9 to `class Counter2` by writing a `toString` method for it.
- We use as the application's model, a `Counter2[]` object, that is, an array of counters.
- We create a view that contains a `JList` whose internal model is exactly the array of counters.
- We add a `Go` button that, when pushed, asks the `JList` which `Counter2` item was selected and then tells that item to increment itself.

Figure 26 shows the model and view classes that generates the example, and Figure 27 shows the controller-button that updates the model.

Figure 10.26: model and view for scrolling list of counters (concl.)

```
import java.awt.*;
import javax.swing.*;
/** ListFrame shows a scrolling list */
public class ListFrame extends JFrame
{ private Counter2[] counters; // the address of the model object
  private JList items;         // the list that displays the model's elements

  /** Constructor  ListFrame generates the frame with the list
    * @param model - the model object that will be displayed as a list */
  public ListFrame(Counter2[] model)
  { counters = model;
    items = new JList(counters);  // embed the model into a JList
    JScrollPane sp = new JScrollPane(items);  // attach a scroll bar
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(2,1));
    cp.add(sp);                   // add the scrolling list to the pane
      JPanel p = new JPanel(new GridLayout(2,1));
      p.add(new ListButton("Go", counters, this));  // see Figure 27
      p.add(new ExitButton("Quit"));                // see Figure 16
    cp.add(p);
    update();  // initialize the view of the list
    setTitle("ListExample");
    setSize(200,200);
    setVisible(true);
  }

  /** getSelection returns which list item is selected by the user
    * @return the element's index, or -1 is no item is selected  */
  public int getSelection()
  { return items.getSelectedIndex(); }

  /** update refreshes the appearance of the list */
  public void update()
  { items.clearSelection(); }  // deselect the selected item in the list
}
```

Figure 10.27: button-controller for scrolling list example

```
import java.awt.event.*;
import javax.swing.*;
/** ListButton implements a button that alters a scrolling list */
public class ListButton extends JButton implements ActionListener
{ private Counter2[] counters; // address of model object
  private ListFrame view;      // address of view object

  /** Constructor ListButton constructs the controller */
  public ListButton(String label, Counter2[] c, ListFrame v)
  { super(label);
    counters = c;
    view = v;
    addActionListener(this);
  }

  /** actionPerformed  handles a button-push event */
  public void actionPerformed(ActionEvent evt)
  { int choice = view.getSelection();  // get selected index number
    if ( choice != -1 )  //  Note: -1 means no item was selected.
       { counters[choice].increment();
         view.update();
       }
  }
}
```

Class `ListFrame` builds its scrolling list, `items`, from the array of `Counter2[]` objects that its constructor method receives. When the frame is made visible, the view part of `items` automatically sends `toString` messages to each of its array elements and it displays the strings that are returned in the list on the display.

`ListFrame` is equipped with two small but important public methods, `getSelection` and `update`. When invoked, the first asks the list for which item, if any, is selected at this time by the user. The second method tells the list to deselect the item so that no list item is selected.

Perhaps the user pushes the `Go` button; the `actionPerformed` method for `ListButton` sends a `getSelection` message to the view and uses the reply to increment the appropriate counter in the model. Then an `update` message deselects the selected item. Once `actionPerformed` finishes, the scrolling list is redrawn on the display, meaning that the `toString` methods of the counters report their new values.

It is possible to attach event listeners directly to the scrolling list, so that each time the user selects an item, an event is generated and an event handler is invoked. A controller that handles such *list selection events* must implement the

ListSelectionListener interface in Figure 7. For example, we can remove the Go button from the frame in Figure 26, delete Figure 27 altogether, and replace the latter with this controller:

```
import javax.swing.*;
import javax.swing.event.*;
/** ListController builds controllers for lists of counters  */
public class ListController implements ListSelectionListener
{ private Counter2[] counters; // address of model object
  private ListFrame view;      // address of view object

  /** Constructor ListController constructs the controller */
  public ListController(Counter2[] c, ListFrame v)
  { counters = c;
    view = v;
  }

  /** valueChanged responds to a list item selection */
  public void valueChanged(ListSelectionEvent e)
  { int choice = view.getSelection();  // get selected index number
    if ( choice != -1 )
       { counters[choice].increment();
         view.update();
       }
  }
}
```

Then, within the constructor method of class ListFrame, we attach the controller as a listener to the JList items:

```
items.addListSelectionListener(new ListController(counters, this));
```

This makes the list's items behave as if they are buttons, all connected to the same controller.

Finally, it is possible to tell a scrolling list to allow simultaneous selection of multiple items;

```
items.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
```

tells list items to allow multiple selections. Of course, when an event listener examines the list, it should ask for all the selected items; this is done by

```
items.getSelectedIndices()
```

which returns as its answer an array of integers.

Exercises

1. Create a GUI that displays a scrolling list whose items are the first 10 letters of the alphabet. (Hint: use as the list's model this array: `String [] letters = {"a", "b", "c", "d", "e", "f", "g", "h", "i", "j"}`.)

2. Augment the GUI from the previous Exercise with a label and a button. When the user selects a list item and pushes the button, the letter on the selected list item is displayed on the label.

3. Augment the GUI from the previous Exercise so that when the user pushes the button, the text in the selected item is "doubled," e.g., if `a` is selected, it becomes `aa`.

4. Create a list that contains the strings, `Red`, `Yellow`, and `Blue`. Insert the list and a button into a GUI, and program the button so that when the user pushes it, the GUI's background is colored with the color selected in the list.

10.9 Text Fields

AWT/Swing provides a `JTextField` component, which lets a user type one line of text into a text field. An example text field appears in Figure 2 at the start of this Chapter. Typing text into a text field generates events (but not action events) that are processed by the default event listener for a text field; the event listener displays the typed text in the text field, and it will accommodate backspacing and cursor movement. One problem that arises with text fields is that a program's user might type something inappropriate into the text field, so the program must be prepared to issue error messages in response.

Creating and adding a text field to a frame is easy, e.g.,

```
JTextField input_text = new JTextField("0", 8);
Container cp = getContentPane();
 ...
cp.add(input_text);
```

The first statement creates a `JTextField` object that displays 8 characters of text and initially shows the string, `"0"`.

The standard operations one does with a text field is extract the text the user has typed, e.g.,

```
String data = input_text.getText();
```

and reset the text in the text field with a new string:

```
input_text.setText("0");
```

These operations might be used when, say, the user pushes a button that starts an action listener that extracts and resets the text field's contents.

As Figure 5 indicates, a `JTextField` is a subclass of a `JTextComponent` and therefore inherits a variety of methods for cutting, copying, pasting, and selecting. We will not study these methods for the moment; they are more useful for so-called *text areas* (multi-line text components), which we study in a later section.

To show use of a text field, we develop a simple temperature convertor, which accepts a numerical temperature, either Fahrenheit or Celsius, and converts it to the equivalent temperature of the other scale. When the user types a temperature into a text field, selects either the Celsius or Fahrenheit scale, and pushes the `Go` button, the result is displayed in the view:

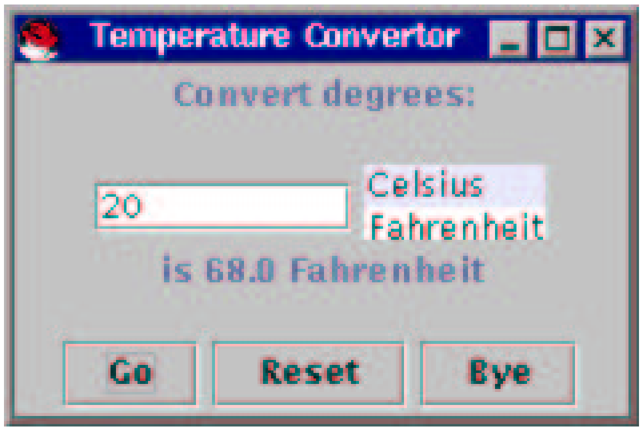


Figure 28 displays the view class for the temperature converter. It is written in two parts: An abstract class provides all the methods but one, and a concrete class extends the abstract one with a coding for `displayError`, a method that displays error messages.

The view's public methods will be used by the controllers, `ResetButton` and `ComputeTempButton`, to fetch the text the user typed, to display the answer that the model computes, and to reset the text field for another conversion. Figure 29 displays the two controllers as well as a simplistic model class, `TempCalculator`.

`ComputeTempButton` uses the model to convert temperatures. For the moment, ignore the `try...catch` exception handler within the button's `actionPerformed` method and examine its interior: The controller sends the view a `getInputs` message to receive an array of two strings, one containing the input temperature and one containing the temperature's scale. The first string is converted into a double. Assuming that the conversion is successful, the scale is examined and used to choose the correct conversion method from the model object. Finally, the view's `displayAnswer` method shows the converted temperature.

What happens if the user types a bad temperature, e.g., `"abc0"`? In this case, the statement,

Figure 10.28: view class for temperature converter

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/** AbsTempFrame creates a view that displays temperature conversions */
public abstract class AbsTempFrame extends JFrame
{ private String START_TEXT = "0";
  private String BLANKS = "              ";

  private JTextField input_text = new JTextField(START_TEXT, 8);
  private JLabel answer = new JLabel(BLANKS);
  // components for the temperature scales list:
  private String[] choices = {"Celsius", "Fahrenheit"};
  private JList scales = new JList(choices);

  /** AbsTempFrame constructs the frame */
  public AbsTempFrame()
  { // the controller that triggers temperature conversion; see Figure 29:
    ComputeTempButton compute_controller = new ComputeTempButton("Go", this);
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(4, 1));
      JPanel p1 = new JPanel(new FlowLayout());
      p1.add(new JLabel("Convert degrees:"));
    cp.add(p1);
      JPanel p2  = new JPanel(new FlowLayout());
      p2.add(input_text);
      p2.add(scales);
    cp.add(p2);
      JPanel p3 = new JPanel(new FlowLayout());
      p3.add(answer);
    cp.add(p3);
      JPanel p4 = new JPanel(new FlowLayout());
      p4.add(compute_controller);
      p4.add(new ResetButton("Reset", this));  // see Figure 29
      p4.add(new ExitButton("Bye"));           // see Figure 16
    cp.add(p4);
    resetFields();  // initialize the view
    setSize(240, 180);
    setTitle("Temperature Convertor");
    setVisible(true);
  }
 ...
```

Figure 10.28: view class for temperature converter (concl.)

```
  /** getInputs returns the inputs the user typed and selected.
    * @return  (1) the string the user typed, and
    *          (2) "Celsius" or "Fahrenheit"  */
  public String[] getInputs()
  { String[] input = new String[2];
    input[0] = input_text.getText();
    input[1] = choices[scales.getSelectedIndex()];
    return input;
  }

  /** displayAnswer resets the label
    * @param s - the string used to reset the label */
  public void displayAnswer(String s)
  { answer.setText(s); }

  /** displayError displays an error message
    * @param s - the message */
  public abstract void displayError(String s);  // will be coded later

  /** resetFields resets the view's text field */
  public void resetFields()
  { input_text.setText(START_TEXT);
    answer.setText(BLANKS);
    scales.setSelectedIndex(0);  // reset scale selection
  }
}

/** TempFrame builds a completed view for a temperature converter */
public class TempFrame extends AbsTempFrame
{
  public TempFrame()
  { super(); }

  public void displayError(String s)
  { displayAnswer("Error: " + s); }  // invoke method in superclass
}
```

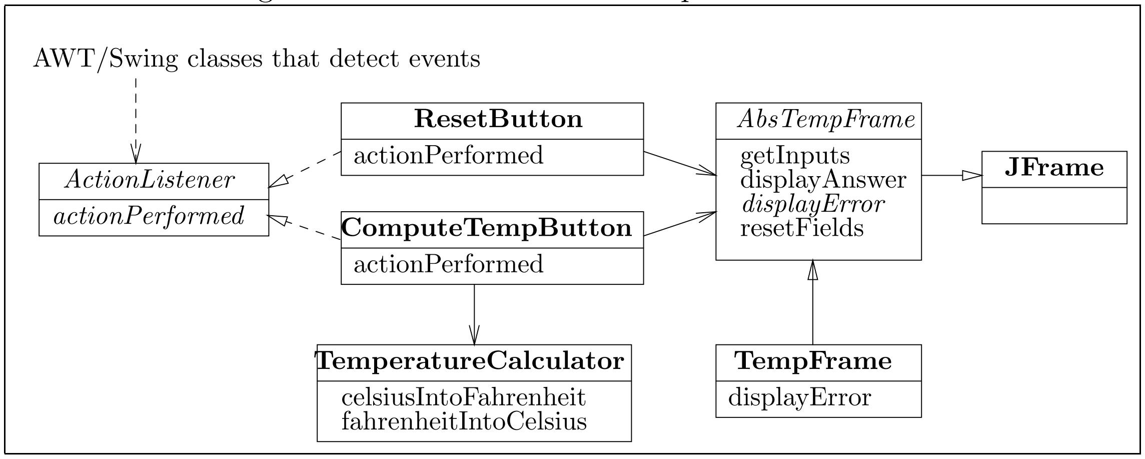
Figure 10.29: controllers and model for temperature conversion

```
import javax.swing.*;
import java.awt.event.*;
/** ComputeTempButton implements a button that converts temperatures */
public class ComputeTempButton extends JButton implements ActionListener
{ private TempCalculator calc = new TempCalculator();
      // the model object for calculating temperatures; see Fig. 5, Ch. 6
  private AbsTempFrame view;  // address of the view object

  /** Constructor ComputeTempButton constructs the button
    * @param v - the address of the view object */
  public ComputeTempButton(String label, AbsTempFrame v)
  { super("Go");
    view = v;
    addActionListener(this);
  }

  /** actionPerformed calculates the temperature */
  public void actionPerformed(ActionEvent evt)
  { try { String[] s = view.getInputs();  // get temp and scale
          double start_temp = new Double(s[0].trim()).doubleValue();
          String answer = "is ";
          if ( s[1].equals("Celsius") )
               { answer = answer + calc.celsiusIntoFahrenheit(start_temp)
                                 + " Fahrenheit"; }
          else { answer = answer + calc.fahrenheitIntoCelsius(start_temp)
                                 + " Celsius"; }
          view.displayAnswer(answer);
        }
    catch(RuntimeException e)   // if s[0] is nonnumeric, an exception occurs
        { view.displayError(e.getMessage()); }
  }
}
```

Figure 10.29: controllers and model for temperature conversion (concl.)

```
import javax.swing.*;
import java.awt.event.*;
/** ResetButton resets the fields of a lottery GUI */
public class ResetButton extends JButton implements ActionListener
{ private AbsTempFrame view;  // address of the view object that gets reset

  /** Constructor ResetButton constructs the button
    * @param v - the address of the view object */
  public ResetButton(String label, AbsTempFrame v)
  { super(label);
    view = v;
    addActionListener(this);
  }

  /** actionPerformed  resets the view's text fields */
  public void actionPerformed(ActionEvent evt)
  { view.resetFields(); }
}

/** TempCalculator models conversion between Celsius and Fahrenheit */
public class TempCalculator
{ /** celsiusIntoFahrenheit translates degrees Celsius into Fahrenheit
    * @param c - the degrees in Celsius
    * @return the equivalent degrees in Fahrenheit */
  public double celsiusIntoFahrenheit(double c)
  { return ((9.0/5.0) * c) + 32; }

  /** fahrenheitIntoCelsius translates degrees Fahrenheit into Celsius
    * @param f - the degrees in Fahrenheit
    * @return the equivalent degrees in Celsius */
  public double fahrenheitIntoCelsius(double f)
  { return (f - 32) * (5.0/9.0); }
}
```

Figure 10.30: architecture of temperature converter



```
double start_temp = new Double(s[0].trim()).doubleValue();
```

cannot complete—it generates a `RuntimeException`. As we learned in Chapter 3, the exception can be caught by the `catch` clause of the exception handler,

```
try {  ...
      double start_temp = new Double(s[0].trim()).doubleValue();
       ...
    }
catch(RuntimeException e)  // if s[0] is nonnumeric, an exception occurs
    { view.displayError(e.getMessage()); }
```

and the statement, `view.displayError(e.getMessage())` executes. This sends a `displayError` message to the view object, where the parameter, `e.getMessage()`, computes to a string that describes the error.

Figure 30 surveys the architecture of the temperature converter. A startup class is needed to create objects from the classes in the Figure. The startup class can be just this:

```
public class Temp
{ public static void main(String[] args)
  { new TempFrame(); }
}
```

Finally, when a user types text into a text field, she often terminates her typing by pressing the *Enter* key. If we desire, we can make the *Enter* key generate an action event; all we need do is add an action listener to the text field object. In Figure 28, the text field was constructed as follows:

```
private JTextField input_text = new JTextField(START_TEXT, 8);
```

and the controller that responds to user input was defined as

```
ComputeTempButton compute_controller = new ComputeTempButton("Go", this);
```

Now, we merely add this statement to the constructor in Figure 28:

```
input_text.addActionListener(compute_controller);
```

This registers the `compute_controller` as the event handler for action events for `input_text`. Now, either a press of the *Enter* key or a click on the `Go` button generates an action event that is handled by `actionPerformed` in `class ComputeTempButton`.

Exercises

1. Create a GUI that contains a text field, a button, and a label. When the user pushes the button, whatever text that appears in the text field is copied to the label, and the text field is reset to hold an empty string.

2. Create a child's arithmetic calculator: Its interface displays two text fields, a label, and a four-item list, where the items are `+`, `-`, `*`, and `/`. When the child types two integers into the two text fields and selects one of the four items, the calculator computes the selected arithmetic operation on the two numbers and displays the result in the label.

3. Revise the temperature converter GUI in Figure 28 so that its `answer` label is removed and is replaced by the output view class, `TemperaturesWriter` in Figure 6, Chapter 5. (Hint: Change its header line to read, `class TemperaturesWriter extends JPanel`, and remove the `setTitle` method from its constructor method. Now, you have a panel you can insert into the `AbsTempFrame` in Figure 28.)

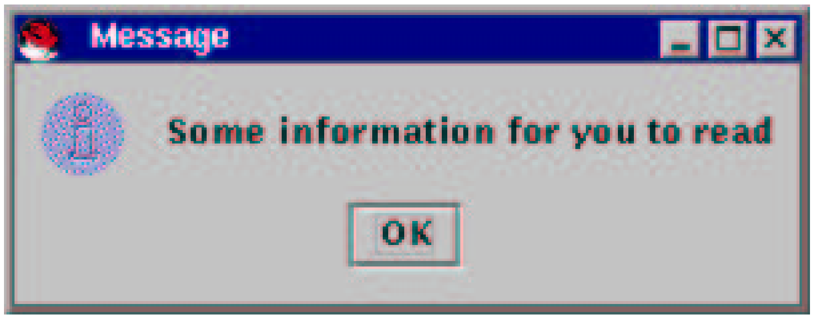
4. Make a GUI that displays a ten-item list, where all ten items are blank. The GUI also has a text field. When the user types a word into the text field and presses *Enter*, the word is copied into the lowest-numbered blank item in the list, if it is not in the list already. (If the word is already in the list or if the list is completely filled, no action is taken.)

10.10 Error Reporting with Dialogs

A graphical user interface usually reports an error by constructing a dialog that displays an error message. A dialog is meant to halt an application's execution and alert its user to a situation that demands immediate attention. After the user responds to the dialog—typically, by pushing one of its buttons—the dialog disappears and the application resumes execution.

Figure 3 at the beginning of the chapter showed the dialog that the temperature converter program might produce when a user enters an invalid temperature for conversion.

As we already know, the AWT/Swing framework contains a `class JOptionPane` that lets one simply generate forms of dialogs. The simplest form of dialog, a *message dialog*,



is created by stating,

```
JOptionPane.showMessageDialog(owner, "Some information for you to read");
```

where `owner` is the address of the frame to which the dialog refers. The `owner` is supplied to the dialog so that the dialog can be displayed near its owner. (If owner is `null`, the dialog appears in the center of the display.)

Execution of the application pauses while the dialog appears on the display; once the user pushes the `OK` button, execution resumes at the position in the program where the dialog was created.

In the previous section, we studied a temperature converter application that displayed error messages in a label in the application's view. The view was constructed from a class, `TempFrame` (see Figure 28), which extended an abstract class, `AbsTempFrame` (Figure 28). We can easily extend `AbsTempFrame` to build a frame that displays error messages within message dialogs. Figure 31 presents `class TempFrame2`, whose `displayError` method generates message dialogs, like the one in Figure 3.

Recall that `displayError` is invoked from the `actionPerformed` method within `class ComputeTempButton` (Figure 29) when a bad number has been typed into the frame's text field. The dialog halts execution of the frame until the user pushes `OK` (or the "X"-button at the dialog's top right corner). We initiate this variant of the temperature converter with this class:

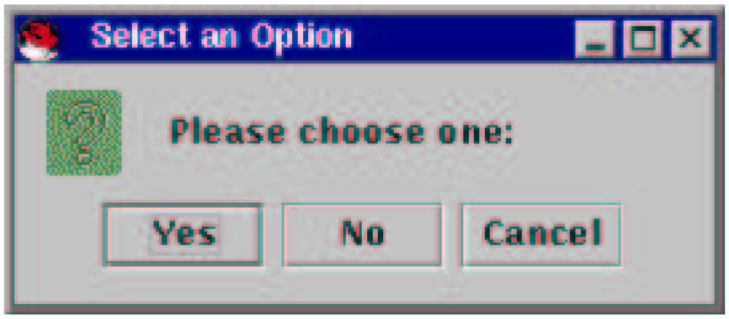
```
public class Temp2
{ public static void main(String[] args)
  { new TempFrame2(); }
}
```

A second form of dialog is the *confirm dialog*, which displays a message and asks

Figure 10.31: frame that generates a message dialog

```
import java.awt.*;
import javax.swing.*;
/** TempFrame2 builds a complete view for a temperature converter that
  *  displays a message dialog in case of an error.  */
public class TempFrame2 extends AbsTempFrame  // see Figure 28
{ public TempFrame2()
  { super(); }

  public void displayError(String s)
  { JOptionPane.showMessageDialog(this, "Error in input: " + s); }
}
```

the user for a decision:



The dialog is generated by a statement like this one:

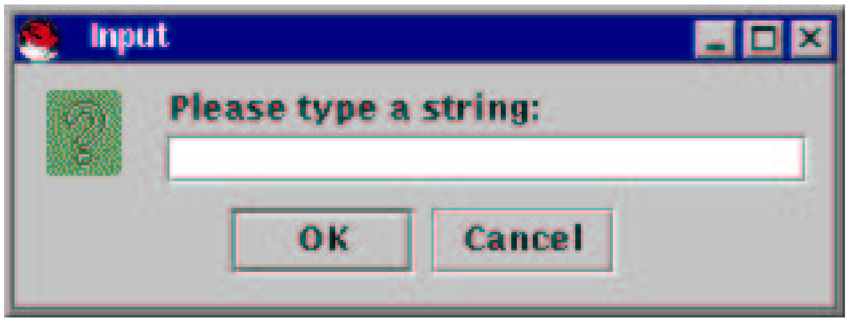
```
int i = JOptionPane.showConfirmDialog(owner, "Please choose one:");
```

Again, `owner` is the address of the component whose execution should be paused while the dialog appears. When the user pushes one of the `Yes`, or `No`, or `Cancel` buttons, an integer value is returned, and in the above case, saved in variable `i`. The value can be queried, e.g.,

```
if ( i == JOptionPane.YES_OPTION )
   { System.out.println("‘Yes’ was pushed"); }
```

The possible values returned are `JOptionPane.YES_OPTION`, `JOptionPane.NO_OPTION`, `JOptionPane.CANCEL_OPTION`, and `JOptionPane.CLOSED_OPTION`, the last arising when the user pushes the “X”-button to terminate the dialog.

The third form of dialog is an *input dialog*, which lets the user type text before dismissing the dialog:



This dialog is created by

```
String s = JOptionPane.showInputDialog(owner, "Please type a string:");
```

When the user types a string into the text field and pushes `OK`, the string is returned, and in this case, assigned to `s`. If the user types nothing but pushes `OK`, the empty string, `""`, is returned. If the user pushes `Cancel`, a `null` value is returned, regardless of what was typed into the text field. Finally, if the user pushes the "X"-button, `null` is returned as well.

Exercises

1. Write a GUI with three buttons and a label. When the user pushes the first button, a message dialog appears; when the user pushes the button on the confirm dialog, the label displays, `Message dialog dismissed`. When the user pushes the second button, a confirm dialog appears; when the user pushes a button on the confirm dialog, the label displays the name of the button pushed. When the user pushes the third button, an input dialog appears; when the user enters text and pushes the dialog's `Ok` button, the label displays the text the user typed.

2. Improve the arithmetic calculator from Exercise 2 of the previous section so that a message dialog is displayed if the user types a nonnumber into one of the calculator's text fields.

10.11 TextAreas and Menus

A *text area* is a text component into which multiple lines of text can be typed; an example of one appears in Figure 4 at the start of the Chapter—the large white area in the frame is the text area. Like buttons, lists, and text fields, a text area has its own little MVC-architecture, where the model part holds the lines of text that the user types into the text area, and the view part displays the model's contents in the large white area. The controller for a text area responds to a user's typing by copying the typed letters into the model part and refreshing the view part so that the user sees the text she typed. The controller also allows the user to move the insertion caret by clicking the mouse and to select text by dragging the mouse.

An easy way to create a text area and embed it into a frame goes as follows:

```
Container cp = getContentPane();
 ...
JTextArea text = new JTextArea("", 20, 40);
text.setLineWrap(true);
text.setFont(new Font("Courier", Font.PLAIN, 14));
JScrollPane sp = new JScrollPane(text);
cp.add(sp);
```

In the above example, the constructor, `JTextArea("", 20, 40)`, creates a text area that displays 20 lines, each line of length 40 columns, where the empty string is displayed initially. (That is, the text area is a big, empty space.) Next, `text.setLineWrap(true)` tells the text area to "wrap" a line of length greater than 40 by spilling the extra characters onto the next line.

If we desire a font different from the text area's default, we state `text.setFont(new Font("Courier", Font.PLAIN, 14))`, where `setFont` sets the font and `new Font(name, style, size)` creates a font in the name, style, and point size that we desire. Standard examples of font names are `"Courier"`, `"TimesRoman"`, and `"SansSerif"`; styles include `Font.PLAIN`, `Font.BOLD`, and `Font.ITALIC`; and sizes between 10 and 16 points are commonly available.

The fourth statement embeds the text area into a scroll bar, so that user input exceeding 20 lines can be scrolled forwards and backwards. Finally, the scrolling text area is embedded into the content pane. The text area in Figure 4 was assembled essentially in this manner.

Text areas possess a wide variety of methods, many of which are inherited from its superclass, `JTextComponent`. Table 32 lists some of the most useful ones. Rather than study the methods in the Table now, we encounter them in the case study in the next section.

A text area's default controller processes typed text, including *Backspace* and *Enter* keys, as expected. It processes mouse movements, clicks, and drags correctly as well. Rather than attach event listeners directly to a text area, we should create *menus* that we hang above the text area. When a user selects an item from a menu, this generates an action event that can signal an event handler that can examine and update the text area.

Menu items are embedded into menus, which are embedded into a menu bar, which is embedded into a frame. For example, the two menus displayed in Figure 4 at Chapter's beginning can be created by these statements inside the constructor method of a frame:

Figure 10.32: methods for text areas

abstract class JTextComponent	
Methods	
`getText(): String`	Return the entire text contents of the component as one string.
`setText(String s)`	Reset the text contents of the components to `s`.
`getCaretPosition(): int`	Return the character position where the insertion caret is positioned.
`setCaretPosition(int p)`	Move the insertion caret to position `p`.
`moveCaretPosition(int p)`	Like `setCaretPosition` but also selects the text that falls between the previous caret position and the new position, `p`.
`getSelectedText(): String`	Return the string that was selected by the user by dragging the mouse across the string.
`getSelectionStart(): int`	Return the index of the first character of the selected text.
`getSelectionEnd(): int`	Return the index of the last character, plus one, of the selected text.
`cut()`	Remove the selected text and hold it in the component's clipboard.
`copy()`	Copy the selected text into the component's clipboard.
`paste()`	Insert a copy of the text in the component's clipboard at the caret position.
`isEditable(): boolean`	Return whether the user may alter the contents of the text component.
`setEditable(boolean b)`	Set whether the user may alter the contents of the text component.

class JTextArea extends JTextComponent	
Methods	
`setFont(Font f)`	Set the font used to display the text to `f`. A typical value for `f` is `new Font("Courier", Font.PLAIN, 14)`.
`setLineWrap(boolean b)`	State whether or not a line longer than the width of the text area will be displayed completely by "wrapping" it to the next line.
`insert(String s, int i)`	Insert string `s` at position `i` in the text area.
`replaceRange(String s, int start, int end)`	Replace the string within the text area starting at position `start` and ending at position `end`-1 by string `s`.

```
JMenuBar mbar = new JMenuBar();
JMenu file = new JMenu("File");    // the  "File"  menu
  ... // statements go here that add menu items to the  File  menu
mbar.add(file);                    // attach menu to menu bar
JMenu edit = new JMenu("Edit");    // the  "Edit"  menu
  // add these menu items to the  Edit  menu:
  edit.add(new JMenuItem("Cut"));
  edit.add(new JMenuItem("Copy"));
  edit.add(new JMenuItem("Paste"));
  edit.addSeparator();             // adds a separator bar to the menu
  JMenu search = new JMenu("Search");
   ...  // statements go here that add menu items to the  Search  menu
  edit.add(search);                // a menu can be added to a menu
mbar.add(edit);
setJMenuBar(mbar);                 // attach menu bar to frame
```

Menu items act like buttons—when selected, they generate action events. As written, the above statements do not attach action listeners to the menu items, but we can do so in the same way that we have done for buttons. We pursue this in the next section.

10.11.1 Case Study: Text Editor

We put menus and text areas to work in an interactive text editor. Figure 4 displays the view of the editor we will build. The editor uses two menus, each of which contains several menu items. A user selects a menu item by clicking the mouse on the menu name, dragging the mouse to the desired item, and releasing the mouse. This generates an action event, like a button push, that can be handled with an `actionPerformed` method. Building menu items, action listeners, and menus will be straightforward.

A first draft of the text editor's architecture appears in Figure 33. The editor's model will be an `EditModel` that extends a `JTextArea`. We do this because the editor's model is just the text contained in the text area—the text area becomes the model. A variety of menu items (controllers) will consult the model and update it. The editor's view will be presented by `EditFrame`, which displays the menus and text area.

When selected, the `ReplaceMenuItem` will display a secondary frame, called `ReplaceFrame`, that helps a user find and replace a string in the text area. Figure 34 shows the `ReplaceFrame` that appears. Unlike a dialog, the `ReplaceFrame` is created when the application starts; it appears and disappears as directed by the user. (Pressing its `Close` button makes the frame disappear.) Unlike a dialog, the frame can be open and operating at the same time the `EditFrame` is also open and operating.

The construction of the view, `class EditFrame`, is portrayed in Figure 35. It is straightforward. The classes for the various menu items and `ReplaceFrame` will be

Figure 10.33: partial class diagram of text editor

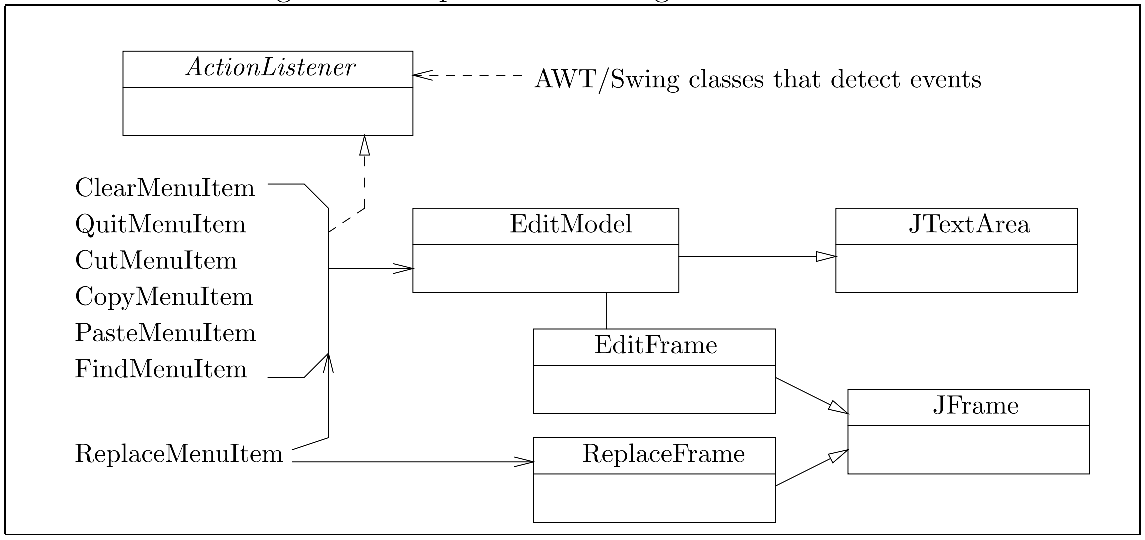
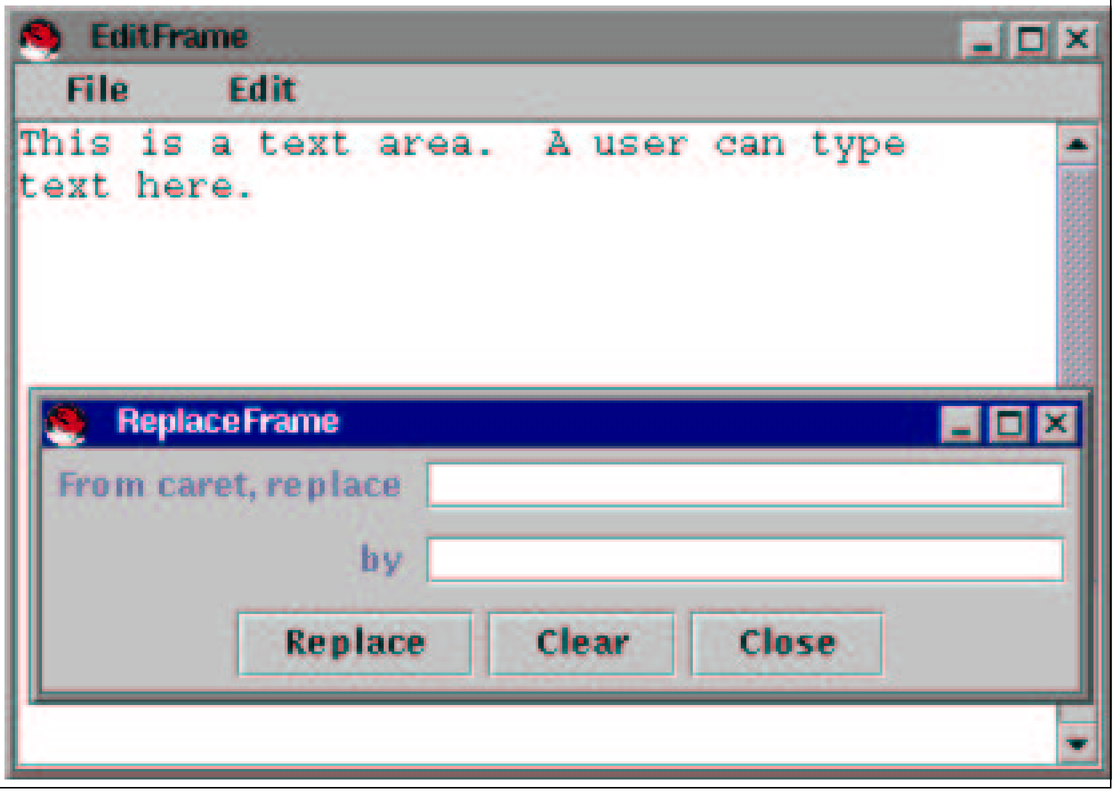


Figure 10.34: editor with auxiliary frame



This is a text area. A user can type
text here.

Figure 10.35: view for text editor

```
import java.awt.*;
import javax.swing.*;
/** EditFrame displays a text editor with two menus and a text area.  */
public class EditFrame extends JFrame
{ // the EditModel, a subclass of JTextArea, is the ‘‘model’’:
  private EditModel buffer = new EditModel("", 15, 50);

  /** Constructor  EditFrame  builds the editor interface */
  public EditFrame()
  { // Create the ReplaceFrame, which appears when the user selects ‘‘Replace’’:
    ReplaceFrame second_frame = new ReplaceFrame(buffer);
    Container cp = getContentPane();
    cp.setLayout(new BorderLayout());
    JMenuBar mbar = new JMenuBar();
      JMenu file = new JMenu("File");     // defines the  "File"  menu
      file.add(new ClearMenuItem("New", buffer));
      file.add(new QuitMenuItem("Exit"));
    mbar.add(file);    // attach menu to menu bar
      JMenu edit = new JMenu("Edit");     // defines the  "Edit"  menu
      edit.add(new CutMenuItem("Cut", buffer));
      edit.add(new CopyMenuItem("Copy", buffer));
      edit.add(new PasteMenuItem("Paste", buffer));
      edit.addSeparator();
        JMenu search = new JMenu("Search");  // defines the "Search" submenu
        search.add(new FindMenuItem("Find", buffer));
        search.add(new ReplaceMenuItem("Replace", second_frame));
      edit.add(search);
    mbar.add(edit);
    setJMenuBar(mbar);  // attach menu bar to frame
    JScrollPane sp = new JScrollPane(buffer);  // embed into a scroll pane
    cp.add(sp, BorderLayout.CENTER);
    setTitle("EditFrame");
    pack();
    setVisible(true);
  }
}
```

seen momentarily. (For the moment, pretend the menu items are like the buttons we have used.)

Next, Figure 36 shows the model/text area, `EditModel`. The model extends a `JTextArea` with methods to clear the text area and to find strings within it. The most interesting method, `find`, fetches the text within the text area with the `getText` method. (See Table 32.) Next, it uses a string search method for strings: `text.indexOf(s, position)` returns the index within `text`, starting from `position`, where string `s` first appears. (If `s` does not appear, -1 is returned.) Next, the caret is moved to the end position where `s` was found (by using `setCaretPosition`) and is dragged backwards, selecting the found string, by `moveCaretPosition`. (See Table 32.)

The model's methods are used by the various menu items, which are listed in Figure 37.

The majority of the menu items do no more than send a single message to `EditModel`; these menu items are written as subclasses of an abstract class, `EditorMenuItem`.

`FindMenuItem` works a bit harder: Its `actionPerformed` method generates an input dialog that asks the user for a string to find. Then, the `EditModel`'s `findFromCaret` method is asked to locate the string. If the string is not found, the user is asked, by means of a confirm dialog, if the search should be performed from the front of the text area. If the user so wishes, this is done.

The last menu item, `ReplaceMenuItem`, displays the `ReplaceFrame`, which helps a user find a string and replace it by another. The `ReplaceFrame` is depicted in Figure 38.

Although it increases the size of the class and limits flexibility, we have made `ReplaceFrame` the action listener for its three buttons. This shows how one `actionPerformed` method can handle three different button pushes in three different ways—the message, `e.getSource()`, asks parameter `e` the identity of the button that was pushed, and based on the answer, the appropriate steps are executed. The method, `replaceRange`, in the text area is used to replace the string that is found by the string the user typed as the replacement; see Table 28.

Exercises

1. Add to the text editor's GUI a menu that lists these font sizes: `12`, `14`, and `16`. When the user selects one of the sizes from the menu, the text displayed in the text area changes to the selected size.

2. You can add buttons to a menu–try this: Modify `class AbsTempFrame` in Figure 28 so that it uses a menu to hold the `Go`, `Reset`, and `Bye` buttons.

3. Create an "appointments" GUI that displays a text area, a `Save` button, and a five-item menu consisting of `Monday`, `Tuesday`, `Wednesday`, `Thursday`, and `Friday`. When the GUI's user selects one of the menu items, the text area displays a message saved for the item. (Initially, all the messages for all items are empty.)

Figure 10.36: text area for editor

```
import java.awt.*;  import javax.swing.*;
/** EditModel models a text area  */
public class EditModel extends JTextArea
{ /** EditModel builds the text area
    * @param initial_text - the starting text for the text area
    * @param rows - the number of rows
    * @param cols - the number of columns  */
  public EditModel(String initial_text, int rows, int cols)
  { super(initial_text, rows, cols);   // create the underlying JTextArea
    setLineWrap(true);
    setFont(new Font("Courier", Font.PLAIN, 14));
  }

  /** clear resets the text area to be empty */
  public void clear()
  { setText(""); }

  /** find locates string  s  in the text area, starting from  position */
  private int find(String s, int position)
  { String text = getText();
    int index = text.indexOf(s, position);       // see Table 9, Chapter 3
    if ( index != -1 )                           // did we find string  s?
         { setCaretPosition(index + s.length()); // resets the caret
           moveCaretPosition(index);             // selects the string
         }
    return index;
  }

  /** findFromStart locates a string starting from the front of the text area
    * @param s - the string to be found
    * @return the position where s is first found; -1, if s not found  */
  public int findFromStart(String s)
  { return find(s, 0); }

 /** findFromCaret locates a string starting from the caret position
    * @param s - the string to be found
    * @return the position where s is first found; -1, if s not found  */
  public int findFromCaret(String s)
  { return find(s, getCaretPosition()); }
}
```

Figure 10.37: menu item-controllers for text editor

```
import javax.swing.*;  import java.awt.event.*;
/** QuitMenuItem terminates the text editor.  */
public class QuitMenuItem extends JMenuItem implements ActionListener
{ public QuitMenuItem(String label)
  { super(label);
    addActionListener(this);
  }

  public void actionPerformed(ActionEvent e)
  { System.exit(0); }
}

import javax.swing.*;  import java.awt.event.*;
/** EditorMenuItem defines a generic menu item for the text editor  */
public abstract class EditorMenuItem extends JMenuItem implements ActionListener
{ private EditModel buffer; // address of the model manipulated by the menu item

  public EditorMenuItem(String label, EditModel model)
  { super(label);
    buffer = model;
    addActionListener(this);
  }

  /** myModel returns the address of the model this menu item manipulates */
  public EditModel myModel()
  { return buffer; }

  public abstract void actionPerformed(ActionEvent e);
}

import java.awt.event.*;
/** ClearMenuItem clears a text area */
public class ClearMenuItem extends EditorMenuItem
{ public ClearMenuItem(String label, EditModel model)
  { super(label, model); }

  public void actionPerformed(ActionEvent e)
  { myModel().clear(); }
}
```

Figure 10.37: menu item-controllers for text editor (cont.)

```
import java.awt.event.*;
/** CutMenuItem  cuts the selected text from the text area. */
public class CutMenuItem extends EditorMenuItem
{ public CutMenuItem(String label, EditModel model)
  { super(label, model); }

  public void actionPerformed(ActionEvent e)
  { myModel().cut(); }
}

import java.awt.event.*;
/** CopyMenuItem  copies selected text into the clipboard */
public class CopyMenuItem extends EditorMenuItem
{ public CopyMenuItem(String label, EditModel model)
  { super(label, model); }

  public void actionPerformed(ActionEvent e)
  { myModel().copy(); }
}

import java.awt.event.*;
/** PasteMenuItem moves contents of the clipboard into the text area */
public class PasteMenuItem extends EditorMenuItem
{ public PasteMenuItem(String label, EditModel model)
  { super(label, model); }

  public void actionPerformed(ActionEvent e)
  { myModel().paste(); }
}
```

Figure 10.37: menu item-controllers for text editor (concl.)

```
import javax.swing.*;  import java.awt.event.*;
/** FindMenuItem generates a dialog to find a string in the text area */
public class FindMenuItem extends EditorMenuItem
{ public FindMenuItem(String label, EditModel model)
  { super(label, model); }

  public void actionPerformed(ActionEvent e)
  { String s = JOptionPane.showInputDialog(this, "Type string to be found:");
    if ( s != null )
       { int index =  myModel().findFromCaret(s);
         if ( index == -1 )
            { int response = JOptionPane.showConfirmDialog(this,
      "String " + s + " not found.  Restart search from beginning of buffer?");
              if ( response == JOptionPane.YES_OPTION )
                 { index = myModel().findFromStart(s);
                   if ( index == -1 )
                      { JOptionPane.showMessageDialog(this,
                                        "String " + s + " not found");
                      }
                 }
            }
       }
  }
}

import javax.swing.*;  import java.awt.event.*;
/** ReplaceMenuItem shows the frame that helps the user replace strings */
public class ReplaceMenuItem extends JMenuItem implements ActionListener
{ private ReplaceFrame my_view;

  public ReplaceMenuItem(String label, ReplaceFrame view)
  { super(label);
    my_view = view;
    addActionListener(this);
  }

  public void actionPerformed(ActionEvent e)
  { my_view.setVisible(true); }
}
```

Figure 10.38: frame that replaces strings

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/** ReplaceFrame shows a frame that helps a user find and replace a string */
public class ReplaceFrame extends JFrame implements ActionListener
{ private EditModel model;
  private JButton replace = new JButton("Replace");
  private JButton clear = new JButton("Clear");
  private JButton close = new JButton("Close");
  private JTextField find_text = new JTextField("", 20);
  private JTextField replace_text = new JTextField("", 20);

  public ReplaceFrame(EditModel my_model)
  { model = my_model;
    Container cp = getContentPane();
    cp.setLayout(new BorderLayout());
    JPanel p1 = new JPanel(new GridLayout(2, 1));
      JPanel p11 = new JPanel(new FlowLayout(FlowLayout.RIGHT));
        p11.add(new JLabel("From caret, replace "));
        p11.add(find_text);
      p1.add(p11);
      JPanel p12 = new JPanel(new FlowLayout(FlowLayout.RIGHT));
        p12.add(new JLabel("by "));
        p12.add(replace_text);
      p1.add(p12);
    cp.add(p1, BorderLayout.CENTER);
    JPanel p2 = new JPanel(new FlowLayout());
      p2.add(replace);
      p2.add(clear);
      p2.add(close);
    cp.add(p2, BorderLayout.SOUTH);
    replace.addActionListener(this);
    clear.addActionListener(this);
    close.addActionListener(this);
    setTitle("ReplaceFrame");
    pack();
    setVisible(false);
  }
```

Figure 10.38: frame that replaces strings (concl.)

```
  /** actionPerformed handles all button pushes on this frame
    * @param e - contains the identity of the button that is pushed */
  public void actionPerformed(ActionEvent e)
  { if ( e.getSource() == close )        // was it the Close button?
       { setVisible(false); }
    else if ( e.getSource() == clear )    // the Clear button?
       { find_text.setText("");
         replace_text.setText("");
       }
    else if ( e.getSource() == replace )  // the Replace button?
       { String find = find_text.getText();
         int location = model.findFromCaret(find);
         if ( location == -1 )            // string not found?
              { JOptionPane.showMessageDialog(this,
                         "String " + find + " not found");
              }
         else { model.replaceRange(replace_text.getText(),
                                   location, location + find.length());
              }
       }
  }
}
```

The user can edit the message, and when she pushes `Save`, the message is saved with the selected item.

4. Create a GUI that displays two text areas. Write controllers that let a user cut, copy, and paste text from one text area to the other.

10.12 Event-Driven Programming with Observers

We saw in this Chapter how events, triggered by button pushes and menu-item selects, direct the execution of a program. We can write a program that triggers its own events internally by using the Java class `Observer` and interface `Observable` from the `java.util` package. This lends itself to an event-driven programming style that further "decouples" components from one another.

First, we must learn how event listeners are programmed into graphical components in AWT/Swing. Every event-generating graphical component (e.g., a button) has, as part of its internal model, an array of addresses of its *listener objects*. A listener object, `ob`, is added to button `b`'s array by the message, `b.addActionListener(ob)`.

Figure 10.39: a counter that generates its own events

```
import java.util.*;
/** Counter3  holds a counter that can be observed by Observers */
public class Counter3 extends Observable
{ private int count;   // the count

  /** Constructor  Counter3  initializes the counter
    * @param start - the starting value for the count  */
  public Counter3(int start)
  { count = start; }

  /** increment  updates the count and signals all Observers  */
  public void increment()
  { count = count + 1;
    setChanged();         // marks that an event has occurred
    notifyObservers(); }  // signals all Observers that an event has occurred

  /** countOf accesses count.
    * @return the value of  count */
  public int countOf()
  { return count; }
}
```

In the general case, a component can have multiple listeners and a listener can be registered with multiple components. (But for simplicity, we usually match components with their listeners on a one-one basis.) When an event occurs within the component, the addresses of all its listeners are fetched, and each listener is sent an `actionPerformed` message.

It is possible to implement listeners and event handling for nongraphical components. An object can be made to generate its own "events," and when an object does this, the object's *Observers* are sent an `update` message, which is asking the Observer to "handle" the event. Object, `ob`, is added to object `b`'s Observers by the message, `b.addObserver(ob)`.

Here is an example. We rewrite `class Counter` in Figure 9 so that it can generate its own events for its Observers. The new class appears in Figure 39. The class begins with `import java.util.*`, and it `extends Observable`; this gives the class two new methods, `setChanged` and `notifyObservers`, which are used in the class's `increment` to generate an event and to signal the class's Observers. The class also inherits another method, `addObserver`, which we see momentarily.

An important aspect of the Figure is that *the counter does not know who its Observers are.* This ensures that the counter is not "coupled" too strongly to the other classes in the program. (See Chapter 6 for a discussion about coupling.)

Next, say that another class, call it `class PrintCount`, wishes to print a message to the display every time the counter is incremented. The class must register itself as one of the Observers of `Counter3`. It can be done like this:

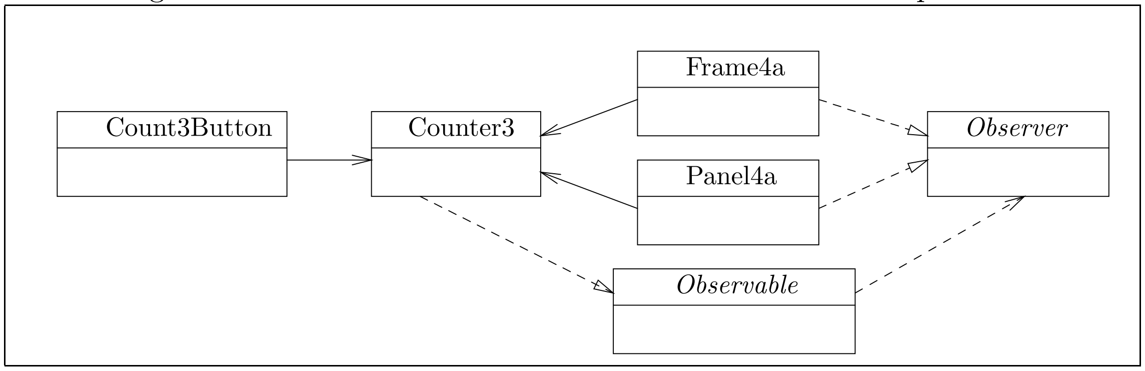
```
import java.util.*;
public class PrintCount implements Observer
{ private Counter3 count;

  public PrintCount(Counter3 c)
  { count = c;
    count.addObserver(this);  // this object registers as an Observer of  count
  }

  /** update prints the newest value of the counter
    * @param ob - the object that signalled the event (here, the counter)
    * @param extra_arg - a parameter that we will not use */
  public void update(Observable ob, Object extra_arg)
  { System.out.println("new count = " + count.countOf()); }
}
```

The class `implements Observer` (see Figure 7), which means that it possesses an `update` method. With the message, `count.addObserver(this)`, the newly created `PrintCount` object registers itself as an Observer of its `Counter3` parameter. Now, whenever the counter object generates a `notifyObservers` message, `PrintCount`'s `update` method will be invoked.

The `Observer` interface requires that `update` possess two parameters; the first contains the address of the object that sent the `notifyObservers` message. (In the above example, this must be the counter object.) The second parameter holds a value, `v`, that is sent when the observed object sends a `notifyObservers(v)` message. (In the above example, the second argument will have value `null`, since the simpler message, `notifyObservers()`, was used.)

As noted above, an advantage to using `class Observable` and `interface Observer` is that it lets one class activate the method of a second class without knowing the identity of the second class. We exploit this in the next section. But a major disadvantage is that the class that generates the events `extends Observable`. Since Java allows a class to extend at most one other class, this makes it impossible to use `class Observable` with any class that already extends another. For example, we cannot revise the scrolling list example in Figure 26 so that `class Counter2` extends `class Observable`, because `Counter2` already extends `class Counter` from Figure 9. (The best we can do is revise the header line of `class Counter2` in Figure 26 so that it `extends Counter3`, producing a slightly convoluted development.)

Figure 10.40: MVC-architecture with Observers and multiple views



10.12.1 Observers and the MVC-Architecture

A primary motivation for writing applications in Model-View-Controller style was to decouple the application's components so that they are easily modified and reused. We have been most successful in this regard with the model components, which are ignorant of the views and controllers that use them, and least successful with the controllers, which know the identities of the model and all the views that depict the model.

Applications with GUIs often have complex views that evolve while the application executes, and it may be unrealistic to make a controller contact all the view objects that must change after the controller sends a message to the model. In this situation, it is better to make the controller ignorant of the views and make the views into Observers of the model. For this reason, we learn how to employ Observers in MVC-architectures.

Figure 19 showed a simple GUI that displayed two "views" of a counter—a numerical view in a frame and a graphical view in a panel. The application that generated the GUI appeared in Figure 20. It used a controller, `class CountButton`, that was given the identity of both the counter and the frame. (See Figure 13.) When the user pushed the `CountButton`, the button sent a message to the counter and a message to the frame. The latter refreshed its numerical count and told the panel to repaint its graphical presentation.

We can simplify the `CountButton` so that it knows nothing about view objects; we also decouple the frame from the panel. We do these steps by registering the frame and the panel as two Observers of the model. The MVC-architecture we design appears in Figure 40. The Figure shows that the controller is decoupled from the view; the views are notified indirectly to update themselves, through the `Observer` interface. And, thanks to `class Observable`, the model remains decoupled from the other components in the application.

Figure 41 presents the controller, `Count3Button`, and Figure 42 shows the view classes, `Frame4a` and `Drawing4a`. Finally, Figure 43 gives the start-up class that creates

Figure 10.41: controller decoupled from views

```
import javax.swing.*;
import java.awt.event.*;
public class Count3Button extends JButton implements ActionListener
{ private Counter3 model;  // see Figure 39

  public Count3Button(String my_label, Counter3 my_model)
  { super(my_label);
    model = my_model;
    addActionListener(this);
  }

  public void actionPerformed(ActionEvent evt)
  { model.increment(); }
}
```

the application's objects and registers them as the model's Observers.

When an application with a GUI is said to have an "MVC-architecture," it normally means that the components are connected as depicted in Figure 40—controllers are decoupled from views, and views independently update themselves when contacted by events generated by the model. Because of Java's demand that a model extend `Observable`, it might not be possible to use this pattern in all cases, however, so our previous MVC-designs still have value.

Exercises

1. For practice, return to Figures 13-15. Replace `class Counter` by `class Counter3` in Figure 39; revise `class Frame2c` to be an Observer of `Counter3`; and simplify `class CountButton` so that it no longer sends a message to the view. Redraw the application's class diagram and compare it to Figure 15; to how many classes is the controller coupled? the model? the view?

2. As in the previous example, convert the slide-puzzle application in Figures 24 and 25 so that its model has Observers.

10.13 Summary

We studied how to design and built graphical user interfaces (GUIs) for applications. Here are the new concepts:

Figure 10.42: view classes

```
import java.awt.*;  import javax.swing.*;  import java.util.*;
/** Frame4a is a frame with a label, button, and panel */
public class Frame4a extends JFrame implements Observer
{ private Counter3 count;  // address of model object; see Figure 39
  private JLabel lab = new JLabel("count = 0");  // label for the frame

  /** Constructor  Frame4a creates a frame with label, drawing, and 2 buttons
    * @param c - the model object, a counter
    * @param drawing - a panel that displays a drawing  */
  public Frame4a(Counter3 c, JPanel drawing)
  { count = c;
    Container cp = getContentPane();
      cp.setLayout(new BorderLayout());
      JPanel p1 = new JPanel(new FlowLayout());
        p1.add(lab);
      cp.add(p1, BorderLayout.NORTH);
      cp.add(drawing, BorderLayout.CENTER);
      JPanel p2 = new JPanel(new FlowLayout());
        p2.add(new Count3Button("Count", count));
        p2.add(new ExitButton("Quit"));
      cp.add(p2, BorderLayout.SOUTH);
    setTitle("Example4");
    setSize(200,150);
    setVisible(true);
  }

  /** update revises the label that displays the count  */
  public void update(Observable model, Object extra_arg)
  { lab.setText("count = " + count.countOf()); }
}
```

Figure 10.42: view classes (concl.)

```
import java.awt.*;  import javax.swing.*;  import java.util.*;
/** Panel4a creates a panel that displays a small painting. */
public class Panel4a extends JPanel implements Observer
{ private Counter3 count;

  public Panel4a(Counter3 model)
  { count = model;
    setSize(200, 80);
  }

  /** update repaints the panel */
  public void update(Observable model, Object extra_arg)
  { repaint(); }

  public void paint(Graphics g)
  { g.setColor(Color.white);
    g.fillRect(0, 0, 150, 80);
    g.setColor(Color.red);
    for ( int i = 0; i != count.countOf(); i = i+1 )
        { g.fillOval(i * 25, 0, 20, 20); }
  }
}
```

Figure 10.43: startup class for application

```
/** Example4a starts the application */
public class Example4a
{ public static void main(String[] args)
  { Counter3 model = new Counter3(0);
    Panel4a panel = new Panel4a(model);
    Frame4a frame = new Frame4a(model, panel);
    model.addObserver(panel);
    model.addObserver(frame);
  }
}
```

New Terminology

- *graphical user interface (GUI)*: an input/output view that lets a user submit input by pressing buttons, selecting menu items, typing text, etc., and shows the user the output in graphical form.
- *event*: input for a program—can be a button push, typing of text, selecting a menu item, etc. The graphical component—button, text field, or menu item—that is used to cause the event is called the *event source*.
- *event-driven programming*: the style of programming one uses to receive and react to events, typically using a collection of controller components, one that does computation in reaction to each form of event.
- *event handler (event listener)*: a controller component that receives and reacts to an event; this is called *handling the event*.
- *action event*: an event caused by a button push or a menu selection. An action event is handled by an event listener called an *action listener*.
- `java.awt` *and* `javax.swing`: the Java packages that contain classes that help a programmer write GUIs. Called *AWT/Swing*, for short.
- *component*: the Java term for a graphical object that has a position, a size, and can have events occur within it.
- *container*: a graphical component that can hold other components. A *panel* is the usual form of container.
- *window*: a "top level" component that can be displayed.
- *frame*: a window with a title bar and menus. Meant to be
- *dialog*: a temporary window that can appear and disappear while an application executes. There are three forms:
  - *message dialog*: displays a warning or error message
  - *confirm dialog*: displays a question that the user must answer by pressing a button
  - *input dialog*: displays a text field into which a user can type input text
- *label*: a component that displays text that the user can read but cannot alter
- *text component*: a component into which a user can type text; this can be a *text field*, into which one line of text is typed, or a *text area*, into which multiple lines can be typed.

- *button*: a component that can be pushed, triggering an action event
- *list*: a component that displays items that can be chosen ("selected")
- *menu*: a component that, when selected, displays a sequence of *menu items*, each of which can be selected, triggering an action event
- *menu bar*: a component that holds a collection of menus
- *layout*: the manner in which a collection of components are arranged for display in a GUI. Some forms of layout are
  - *flow layout*: the components are arranged in a linear order, like the words in a line of text
  - *border layout*: components are explicitly assigned to the "north," "south," "east," "west," or "center" regions of the container
  - *grid layout*: components are arranged as equally-sized items in rows and columns, like the entries of a matrix or grid
- *content pane*: the part of a frame where components are inserted for display
- *glass pane*: the part of a frame that "overlays" the content pane. Normally left "clear" but it can be painted upon by means of a frame's `paint` method.
- *list selection event*: a form of event caused by selecting an item from a list. Such events are handled by *list selection listeners*.

Points to Remember

Some principles to remember are

- A programmer assembles a GUI as a composite of graphical components like buttons, labels, and text fields. The components must be inserted into a frame's *content pane*, which must be told to use a particular *layout* for arranging the components.
- Graphical components can generate *events*, e.g., button pushes or text insertions. Therefore, an application becomes *event driven*—the events generated by button pushes and text insertions, activate controllers (*event handlers* or *event listeners*) that send messages to the application's model to compute results.
- The graphical components in Java's AWT/Swing framework have internal Model-View-Controller (MVC) architectures, and a programmer often works with a a graphical component by reading and updating its internal model. Indeed, for components like text areas and lists, the component sometimes serves as the

model for the overall application—one example is a text editor program, whose model is a text area component.

- The standard MVC architecture for an application with a GUI uses the *observer pattern* to connect its components: events from the GUI trigger controllers. A controller sends a message to the model to compute. The model computes answers and signals its Observers (the view) that new results await display. The Observers fetch the results from the model and display them on the GUI. This pattern decouples controllers from views and encourages better reuse of controller components.

New Classes for Later Use

Figure 5 lists the classes we use for constructing graphical user interfaces.

10.14 Programming Projects

1. Revise the bank-account manager application in Figure 9, Chapter 6, so that its input- and output-view classes are combined into one pleasant-to-use GUI.

2. For any Programming Project that you worked in Chapter 6, write a GUI for it. For example, if you built Project 2, Chapter 6 (the mortgage calculator), then write a helpful GUI into which a user can type principal, interest, and year values and receive answers regarding monthly payments and totals.

3. For any Programming Project that you worked in Chapter 7, write a GUI for it. For example, if you built Project 3, Chapter 7 (the lottery-odds calculator), then write a helpful GUI into which a user can type her lottery picks, get the odds of winning, and then generate a sample play of the lottery.

4. Write an application that lets two humans play tic-tac-toe (noughts-and-crosses) by clicking on the buttons of a 3-by-3 grid.

5. Write an application with a GUI that lets two humans play checkers by clicking on the pieces, which are positioned on an 8-by-8 grid.

6. Build an application that looks and behaves like a typical hand-held calculator, with a grid of buttons for numerals and arithmetic operations and a display at the top that displays the answers of the calculations.

7. Build an appointments-manager application. The application lets a user store, edit, and view appointments that are saved according to a particular day of the month and a particular hour of the day. The application might behave as follows. Initially, the days of the current month are displayed, on buttons:

```
-----------------------------------
|    |  1 |  2 |  3 |  4 |  5 |  6 |
 ----------------------------------
|  7 |  8 |  9 | 10 | 11 | 12 | 13 |
 ---------------------------------
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
 ----------------------------------
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
 ---------------------------------
| 28 | 29 | 30 | 31 |    |    |    |
 ---------------------------------
```

When a user moves the mouse over a date and clicks, a new window appears that displays a summary of the appointments for the selected date. For example, if the 8th is selected, we might see the following window appear:

```
 -----------------------------
| Appointments for the 8th:
|  ----------------------
| | 8:30 French 111 class.
| | 10:30 Calculus class.
| | 12:00 Lunch with Fred.
|  -----------------------
 -----------------------------
```

In addition, the above window must have buttons or menus or text fields that let a user view and edit the details of an appointment, add an appointment, delete an appointment, etc. For example, one might wish to view the details of the 8:30 appointment on the 8th. By somehow selecting and clicking, a new window (or a text area within the existing window) might display:

```
  -------------------------------------
 |  8:30 French 111 class.
 |  Exam followed by video for Lesson 12;
 |  remember to study!
  --------------------------------------
```

The above window might have additional buttons or menus that let one modify, save, or even delete the appointment.

8. Build an application that indexes a user's book or music collection. The application helps a user organize her book listings by genre (e.g., novel, mystery, biography), by author, and by title. A user can enter new books and can query the listing of books by genre, by author, or by title.

When you design the application's GUI, consider the merits and drawbacks of having the GUI be one large window, whose buttons and menus perform all the actions listed above, versus a collection of smaller, specialized windows, which can appear and disappear as needed to perform an operation.

9. Design and build a "pebble dropping game," where a human alternates with a computer at dropping pebbles into six vertical chutes. Like tic-tac-toe, the objective is to be the first player to arrange pebbles in a row, vertically, horizontally, or diagonally. The game board first appears with six empty chutes:

```
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
 -----------
```

The human selects a chute and drops a pebble in it; the pebble falls as far as it can:

```
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | |X| | | |
 -----------
```

The computer responds with its own move:

```
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | |X|O| | |
 -----------
```

The human might respond by dropping a pebble on top of another:

```
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | |X| | |
| | |X|O| | |
 -----------
```

The game continues in this way until a player wins or all chutes are full.

10. Build an animated version of the pebble dropping game, called “Tetris”: Given a playing area 6 columns wide and 12 rows high, the computer generates 4-by-4 blocks that appear at random positions above the playing area and fall to the bottom. As they fall, the human player can operate controls that move a falling block to the left or right. The objective is to evenly stack the falling blocks in the playing area so that the maximum number of blocks can be saved. The game ends when the pile of blocks reaches the top of the playing area.

    (a) Build the basic Tetris game described above, and add a score counter that remembers the number of blocks stacked until the game ends.

    (b) Extend the game so that when a row is completely occupied with blocks, that row disappears from the playing area, and the rows of blocks above it drop down one row.

    (c) Extend the game so that in addition to 4-by-4 blocks,

    ```
     - -
    | | |
     - -
    | | |
     - -
    ```

    the computer drops these 1-by-4 and 2-by-3 configurations:

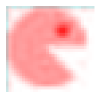
    ```
     - - - -     - -
    | | | | |   | | |
     - - - -     - - -
                  | | |
                   - -
    ```

    When the computer initially drops the two new shapes, it can randomly rotate them, e.g., the 1-by-4 shape might be rotated 90 degrees so that it looks like a 4-by-1 shape. Add controls that rotate the falling shapes clockwise and counterclockwise.

    (d) Modify the game so that the blocks fall downwards faster as a player’s score increases.

11. Build an animated berry-eating game, called “Pac-Man”: The game board is a 12-by-12 grid. Initially, one game square is occupied by the Pac-Man; the

others hold "berries." (It is traditional that the Pac-Man be represented by a smile-face that is ready to "eat," e.g.,



When the game starts, the Pac-Man moves forwards from square to square, eating each berry it finds on a square that it occupies. The human must tell the Pac-Man when to change direction, e.g., turn left or turn right.

(a) Build the basic Pac-Man game described above; attach a clock that counts the elapsed time until the Pac-Man eats all the berries on the board.

(b) Modify the game board so that it is a maze; that is, some squares are occupied by barriers or separated by barriers.

(c) Extend the game so that, approximately every 5 seconds, 3 "goblins" appear on the game board at random positions. Moving at the same speed as the Pac-Man, each goblin tries to "eat" the Pac-Man. If a goblin succeeds, the game ends. The goblins disappear after 5 seconds.

12. Build an animated chase game, called "Frogger": The playing area consists of 6 rows, which are traffic lanes upon which animated cars and trucks travel. The game's player is represented by a "frog," whose objective is to move across all 6 rows without being hit by a moving vehicle. (That is, a vehicle "hits" the frog if it occupies the same playing space as the frog.)

    (a) Build an initial version of the game where there are no vehicles. When the game starts, the frog appears at a random space in the bottommost row of the board. With the use of controls, a player can move the frog forwards or turn the frog left or right.

    (b) Next, create one vehicle for each traffic lane. Each vehicle traverses its lane, and after a short random time delay, reappears to traverse its lane again. Stop the game if a vehicle and the frog land in the same playing space.

    (c) Modify the game so that a seventh row is placed in the middle of the playing area. No vehicles travel on this "divider," but the frog can move onto this space and "rest."

    (d) Modify the game so that vehicles randomly travel at different velocities.

10.15 Beyond the Basics

10.15.1 Applets

10.15.2 Tables and Spreadsheets

10.15.3 Handling Mouse Clicks and Drags

10.15.4 Threads of Execution

10.15.5 GUI Design and Use-Cases

10.15.6 Summary of Methods for Graphical Components

These optional sections described additional capabilities of the AWT/Swing framework. The final section is a tabular summary of all the classes and methods encountered in the chapter.

10.15.1 Applets

At the end of Chapter 4, we saw how to convert an application into an applet. The same technique can be applied, without change, to convert a GUI-based application into an applet: The applet is constructed from the application's view class, so that instead of extending `JFrame` the view class extends `JApplet`. We follow the following steps:

- Remove the `setSize`, `setTitle`, and `setVisible` statements from the frame's constructor method. Also, remove the `addWindowListener` statement, if any.
- Move the instructions in the application's `main` method into the frame's constructor method, and rename the constructor method `public void init()`.
- Write an HTML file for a Web page that starts the applet. A sample file might look like this:

```
&lt;title>My Slide Puzzle&lt;/title>
&lt;body bgcolor=white>
Here is my slide puzzle as an applet:
&lt;p>
  &lt;applet code = "PuzzleApplet.class" width=300 height=300>
  Comments about the applet go here.  This applet will be a simple
  adaptation of  class PuzzleFrame  in Figure 24. &lt;/applet>
&lt;p>
&lt;/body>
```

Once these changes are made, test the applet with the `appletviewer` program (which is available as part of your IDE or as part of the JDK; see Chapter 4), and once the applet meets with your satisfaction, only then use a web browser to see the final result. What you will see is the program's interface embedded into the Web page at the position where you placed the `applet` command.

For example, the GUI in Figure 24 for the slide puzzle is converted into an applet, `PuzzleApplet`, by making the above-listed changes; Figure 44 shows the result.

The other model and view classes stay the same—the program can create dialogs and frames, use lists, menus, and buttons, just like before. One pragmatic disadvantage is that older models of Web browsers might not work correctly with all the AWT/Swing components. This problem should disappear quickly as time passes and proper Web browsers appear.

As seen above, the `applet` command in the HTML file sets the size of an applet and automatically displays it. It is also possible for the HTML file to transmit actual parameters into an applet. For example, perhaps the web page that displays the slide puzzle sets the puzzle's size. This can be done with the following two modifications:

1. Modify the `applet` command in the HTML page to use a `parameter` command:

   ```
   &lt;applet code = "PuzzleApplet.class" width=300 height=300>
   &lt;param name="size" value="5">
   Comments about the applet go here.  This applet will be a simple
   adaptation of class PuzzleFrame in Figure 24. &lt;/applet>
   ```

2. Within the applet's `init` method, replace the statement, `size = 4`, by the following:

   ```
   String s = getParameter("size");
   size = new Integer(s).intValue();
   ```

The HTML `parameter` command creates a `String`-typed "actual parameter" that is fetched by the `getParameter` method inside the applet's `init` method. Multiple parameters can be created this way, as long as each parameter is labelled uniquely in the HTML file.

Finally, it is also possible to "detach" an applet from the web page that creates it so that the applet's GUI appears as a new application on the user's display. Indeed, this is also the easiest way to convert an application into an applet, because only the header line of the `main` method changes; *everything else remains the same*:

- Replace the header line of the `main` method by `public void init()` in the following class:

  ```
  import javax.swing.*;
  public class ExampleApplet extends JApplet
  {
    public void init()
    { ... the body of the  main  method goes here ... }
  }
  ```

Figure 10.44: slide puzzle applet

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/** PuzzleApplet shows a slide puzzle, adapted from Figure 24 */
public class PuzzleApplet extends JApplet
{ private SlidePuzzleBoard board;   // the board that is displayed
  private int size;                 // the board's size
  private int button_size = 60;     // width/height of button
  private PuzzleButton[][] button;  // buttons for the positions on the board

  /** init  builds the interface */
  public void init()
  { // these statements are moved here from the former  main  method:
    size = 4;  // a  4 x 4  slide puzzle
    board = new SlidePuzzleBoard(size); // model
    // the remainder of the constructor method stays the same:
    button = new PuzzleButton[size][size];
    Container cp = getContentPane();
     ...
    // but remember to remove these statements:
    //   addWindowListener(new ExitController());
    //   setTitle("PuzzleFrame");
    //   setSize(size * button_size + 10,  size * button_size + 20);
    //   setVisible(true);
  }

  /** update  consults the model and redraws the puzzle grid. */
  public void update()
  { ... }  // as before; see Figure 24
}
```

- Within the HTML file, use this command:

```
<applet code = "ExampleApplet.class" width=0 height=0>
  This applet displays nothing within the Web page itself, but it generates
  a view just like an ordinary application would do. </applet>
```

For example, if we return to the slide-puzzle example in Figure 24, we can leave `class PuzzleFrame` and all its collaborators untouched; we merely revise the `main` method in the startup class, `Puzzle`, into the following:

```
import javax.swing.*;
public class PuzzleGeneratorApplet extends JApplet
{ public void init()
  { int size = 4;  // a  4 x 4  slide puzzle
    SlidePuzzleBoard board = new SlidePuzzleBoard(size);
    PuzzleFrame frame = new PuzzleFrame(size, board);
  }
}
```

If we start the `PuzzleGeneratorApplet` from this HTML file:

```
<title>Slide Puzzle Generator</title>
<body bgcolor=white>
Congratulations!  You have just started the Slide Puzzle application!
  <applet code = "PuzzleGeneratorApplet.class" width=0 height=0>
  </applet>
</body>
```

we will see the greeting in the web browser and a separate, newly created slide puzzle. The slide puzzle will execute as long as the web browser itself executes.

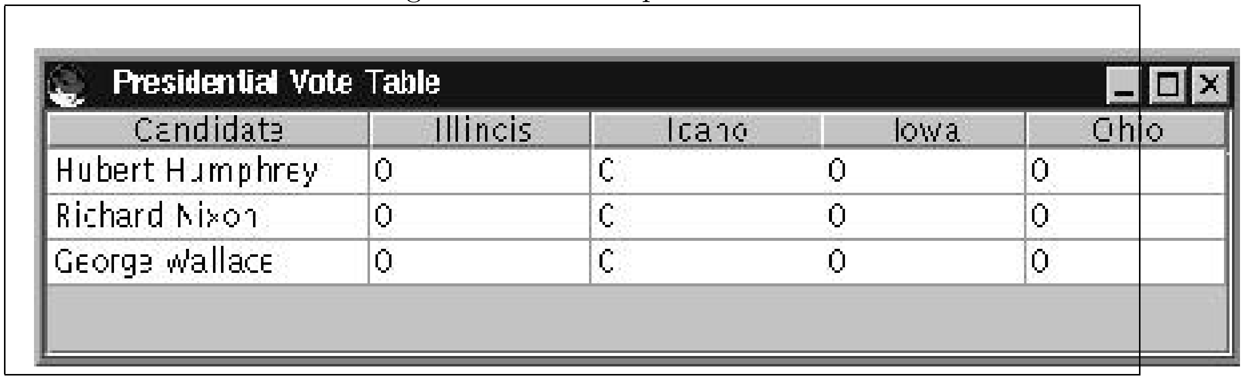
10.15.2 Tables and Spreadsheets

The AWT/Swing framework contains a `class JTable` that makes relatively quick work of creating and displaying tables and spreadsheets. Figure 45 displays a simple GUI that contains a `JTable` component.

The view shows that a table is a grid, containing rows and columns. Each column has its own label. Indeed, the table's view lets a user "stretch" a column's width by moving the mouse to the right edge of the column's label and dragging the label's right edge to the desired width. This trick is helpful for reading extra long strings that might be displayed in the column's cells.

The table's rows are numbered 0, 1, 2, etc.; the labels above the columns are *not* included in the row numbering. For example, in Figure 45 the string, `George Wallace`, appears in the cell at Row 2, Column 0; the label, `Illinois`, appears above Column 1.

Figure 10.45: a simple JTable



Candidate	Illinois	Idaho	Iowa	Ohio
Hubert Humphrey	0	0	0	0
Richard Nixon	0	0	0	0
George Wallace	0	0	0	0

Like other graphical components, a `JTable` has its own internal view, controller, and model. All three can be customized, but we spend our efforts on the model, which is the the part that a programmer must build for her `JTable`. We call the model a *table model*. A table model must be built from a class that implements `interface TableModel`, which is displayed in Figure 46. The interface lists behaviors that a table model must have so that it can be read, updated, and displayed by the controller part and the view part of a `JTable`. We study the interface's methods in the examples that follow.

The best way of building a class that implements `TableModel` is to extend an AWT/Swing prebuilt abstract class, `AbstractTableModel`. This class has codings for all the methods listed in Figure 46, except for `getRowCount`, `getColumnCount`, and `getValueAt`. So, a programmer can quickly construct a table model by extending `AbstractTableModel` with the missing methods.

Figure 47 shows the table model built this way for the view in Figure 45.

The class, `VoteModel`, extends `AbstractTableModel`. It uses three arrays to remember the candidate names that should appear in Column 0, the region names that should appear above the columns, and the vote counts for each candidate in each region. The class implements the missing three methods and revises `getColumnName` so that the table has useful labels for the columns. (Without the last method, the table model would be displayed with labels like `A`, `B`, and `C`.)

Since the table model will be used to count votes in an election, two additional methods, `getVoteCount` and `changeVoteCount`, are inserted. The latter invokes an `AbstractTableModel` method, `fireTableDataChanged`, to signal the table model's listeners when the table is changed; this causes the model's view to update itself on the display.

The crucial method in Figure 47, is `getValueAt`, which the `JTable`'s view repeatedly invokes to learn the contents of the cells in the grid it displays. For simplicity, `getValueAt` can return a result of any object type whatsoever. The method is written so that it returns candidates' names, which are string objects, when asked about Col-

Figure 10.46: interface TableModel

```
public interface TableModel
{ /** getColumnCount  returns the number of columns in the table */
  public int getColumnCount();

  /** getColumnName  returns the label for Column  j  of the table */
  public String getColumnName(int j);

  /** getRowCount  returns the number of rows in the table */
  public int getRowCount() ;

  /** getValueAt  returns an object that contains the value at Row i,
       Column j, within the table */
  public Object getValueAt(int i, int j);

  /** setValueAt  updates Row i, Column j, of the table to have value v */
  public void setValueAt(v, i, j);

  /** isCellEditable returns whether the user can change the value at Row i,
       Column j by editing the display of that cell in the table's view */
  public boolean isCellEditable(int i, int j);

  /** addTableModelListener  adds a listener to the table */
  public void addTableModelListener(TableModelListener l) ;

  /** removeTableModelListener  removes a listener from the table */
  public void removeTableModelListener(TableModelListener l)

  /** getColumnClass  returns the data type of the objects held in Column j */
  public Class getColumnClass(int j);
```

Figure 10.47: model for vote table

```
import javax.swing.table.*;
public class VoteModel extends AbstractTableModel
{ // The table will be displayed with the region names across the top.
  //  Candidate i's name appears at table cell  Row i, Column 0;
  //  Candidate i's votes from Region j appear at table cell  Row i, Column j+1
  private String[] region;     // regions' names
  private String[] candidate;  // candidates' names
  private int[][] votes;       // the votes for candidate i, region j,

  public VoteModel(String[] person, String[] district)
  { super();
    candidate = person;
    region = district;
    votes = new int[candidate.length][region.length];
  }

  public int getRowCount()
  { return candidate.length; }

  public int getColumnCount()
  { return region.length + 1; }

  public Object getValueAt(int table_row, int table_column)
  { Object result = null;
    if ( table_column == 0 ) // is it a the candidate's name?
         { result = candidate[table_row]; }
    else { result = new Integer(votes[table_row][table_column-1]); }
    return result;
  }

  public String getColumnName(int column)
  { String answer = "";
    if ( column == 0 )
         { answer = "Candidate"; }
    else { answer = region[column-1]; }
    return answer;
  }

  /** getVoteCount returns the votes for  person  in  district */
  public int getVoteCount(int person, int district)
  { return votes[person][district]; }
...
```

Figure 10.47: model for vote table (concl.)

```
  /** changeVoteCount  updates  person's  votes in  district  to  new_count
    * and signals the table's listeners that the table has changed  */
  public void changeVoteCount(int person, int district, int new_count)
  { votes[person][district] = new_count;
    fireTableDataChanged();
  }
}
```

umn 0; when entries in other columns are requested, the integer vote count is fetched and "wrapped" in a newly created `Integer` object. Think of `getValueAt` as a clever "array look-up operation" that knows when to return names and when to return vote counts to help the view display the table in Figure 45.

The table model is embedded in a `JTable` object and displayed by the application's frame, which is displayed along with the start-up class, in Figure 48.

Updating the Table

The application constructed in Figure 48 is useless, because there is no means of casting votes and updating the table model. So, we improve the application so that a user can vote from a region for a candidate, and we improve the table model so that it remembers both the votes cast as well as candidates' and regions' totals. The revised table model will be displayed as seen in Figure 49, and a secondary frame will appear in which a user can select one region and one candidate and cast one vote.

The Figure shows that six votes have already been cast in the election, and the seventh vote is ready to be entered.

The hardest work in this application is extending `VoteModel` in Figure 47 so that its `getValueAt` method can correctly calculate the total of a row or the total of a column when asked. Figure 50 shows the result, called `class VoteModelWithTotals`.

The `getValueAt` must handle requests for the values in the extra row and the extra column; all other requests it sends to the `getValueAt` method in its superclass. Notice that `getColumnName` is extended in a simple way to return the label for the extra column.

Figure 51 shows the view class for casting votes, and Figure 52 includes the controller and start-up class for the application.

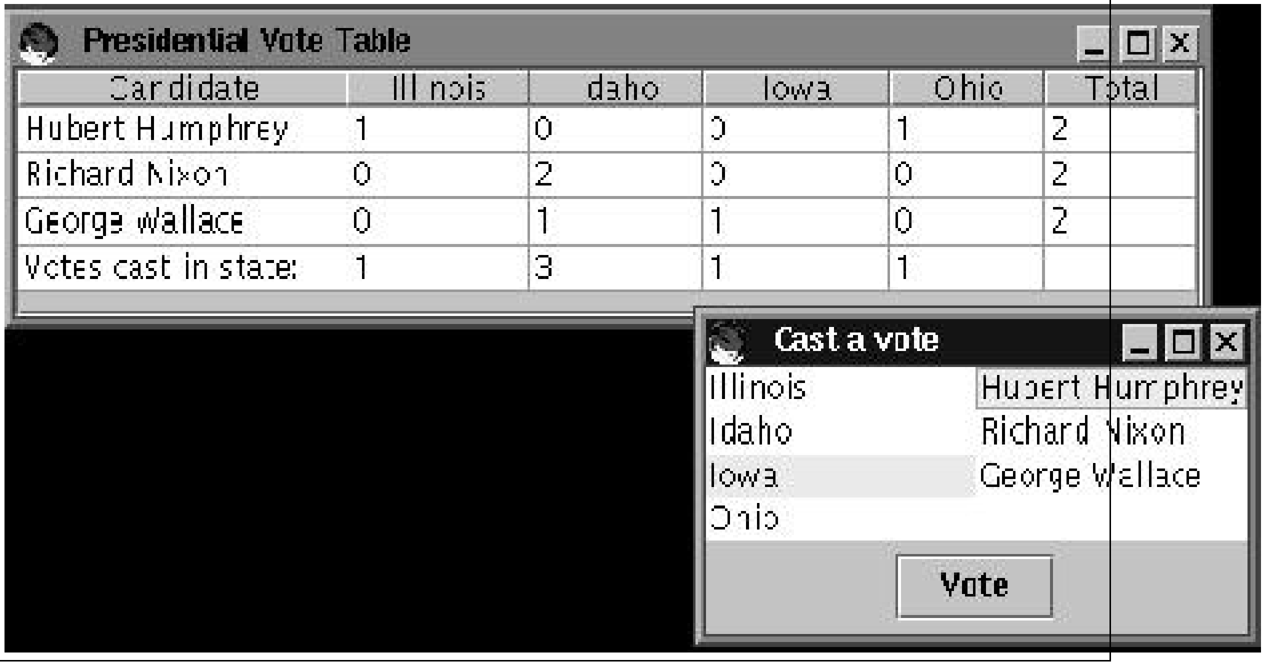
Spreadsheets

For our purposes, a *spreadsheet* is a table that a user can edit by typing new information into the cells displayed in the table's view. A `JTable` can be made into a

Figure 10.48: view for displaying the vote-count table

```
import java.awt.*;
import javax.swing.*;
/** VoteFrame displays a table of votes */
public class VoteFrame extends JFrame
{
  /** VoteFrame constructs the view
    * @param model - a table model  */
  public VoteFrame(VoteModel model)
  { JTable vote_table = new JTable(model); // embed the table model in a JTable
    JScrollPane pane = new JScrollPane(vote_table);
    Container cp = getContentPane();
      cp.setLayout(new BorderLayout());
      cp.add(pane, BorderLayout.CENTER);
    setSize(500, 120);
    setTitle("Presidential Vote Table");
    setVisible(true);
  }
}

/** Vote1 starts the application */
public class Vote1
{ public static void main(String[] args)
  { String[] candidates = {"Hubert Humphrey", "Richard Nixon",
                           "George Wallace"};
    String[] states = {"Illinois", "Idaho", "Iowa", "Ohio"};
    VoteModel model = new VoteModel(candidates, states);
    VoteFrame view = new VoteFrame(model);
  }
}
```

Figure 10.49: vote table with totals and voting frame



Candidate	Illinois	Idaho	Iowa	Ohio	Total
Hubert Humphrey	1	0	0	1	2
Richard Nixon	0	2	0	0	2
George Wallace	0	1	1	0	2
Votes cast in state:	1	3	1	1	

Illinois	Hubert Humphrey
Idaho	Richard Nixon
Iowa	George Wallace
Ohio	

spreadsheet by improving its table model so that the model's cells are "editable." This is done by writing new versions of the table model's `isCellEditable` and `setValueAt` methods.

For example, we can alter the vote table in Figure 49 so that a user can click on a displayed cell and type a new number into it. When the user presses the *Enter* key or clicks on a different cell, this causes the typed valued to replace the one that formerly appeared in the cell. Figure 53 shows how a user is altering Row 2, Column 2 of the table with a vote count of 1000; when the user presses *Enter*, this inserts the 1000 votes and all totals are revised accordingly.

Figure 54 shows how the table model is extended with methods that allow cell editing. The `isCellEditable` method calculates exactly which cells of the table model can be edited. (The cells that hold candidates' names and the cells that display the totals *cannot* be edited.) The `setValueAt` method takes the string that the user typed, attempts to convert it into an integer and update the cell where the user typed it. Badly formed integers are "trapped" by the exception handler.

10.15.3 Handling Mouse Clicks and Drags

AWT/Swing provides a basic collection of *mouse events* that can be handled:

- When a mouse is moved into a component (e.g., a frame or a panel, or a text

Figure 10.50: table model that computes vote totals

```
/** VoteModelWithTotals is a table model that computes votes and totals */
public class VoteModelWithTotals extends VoteModel
{ // The expanded model has one extra column (the total votes for each
  //  candidate) and one extra row (the total votes cast in each region)
  private int number_of_candidates;
  private int number_of_regions;

  public VoteModelWithTotals(String[] candidate, String[] region)
  { super(candidate, region);
    number_of_candidates = candidate.length;
    number_of_regions = region.length;
  }

  public int getRowCount()
  { return super.getRowCount() + 1; }

  public int getColumnCount()
  { return super.getColumnCount() + 1; }

  public Object getValueAt(int table_row, int table_column)
  { Object result = null;
    if ( table_column == (number_of_regions + 1) ) // a candidate's vote total?
         { if ( table_row < number_of_candidates )
                { result = new Integer(computeTotalForPerson(table_row)); }
           else { result = ""; }
         }
    else if ( table_row == number_of_candidates ) // total votes in a region?
         { if ( table_column > 0 )
                { result = new Integer(computeTotalForRegion(table_column-1)); }
           else { result = "Votes cast in state:"; }
         }
    else { result = super.getValueAt(table_row, table_column); }
    return result;
  }

  public String getColumnName(int column)
  { String answer = "";
    if ( column == (number_of_regions + 1) )  // is it the last column?
         { answer = "Total"; }
    else { answer = super.getColumnName(column); }
    return answer;
  }
 ...
```

Figure 10.50: table model that computes vote totals (concl.)

```
  /** computeTotalForPerson  totals all votes for candidate  who */
  private int computeTotalForPerson(int who)
  { int total = 0;
    for ( int j = 0;  j != number_of_regions;  j = j+1 )
        { total = total + super.getVoteCount(who, j); }
    return total;
  }

  /** computeTotalForRegion  totals all votes cast in region  where */
  private int computeTotalForRegion(int where)
  { int total = 0;
    for ( int i = 0;  i != number_of_candidates;  i = i+1 )
        { total = total + super.getVoteCount(i, where); }
    return total;
  }
}
```

area), we say that the mouse *enters* the component. Similarly, the mouse *exits* the component when it is moved outside it.

- When a mouse's button is pushed without moving the mouse, we say that the mouse is *clicked.* It is possible to click the mouse multiple times at one click; a *double click* is an example.
- When a mouse's button is pushed and the mouse is moved while the button is held, we say that the mouse is *pressed.* Eventually, the button is *released.*

Figure 55 lists `interface MouseListener`, which a class must implement to handle the mouse events listed above. The methods of a mouse listener react to mouse events. Table 56 lists some useful methods for extracting information from a mouse event.

A class that `implements MouseListener` might not wish to handle all forms of mouse events. There is a default listener, called `MouseAdapter`, which `implements MouseListener` with "do nothing" event handlers for all mouse events. The simplest way to build a mouse listener with limited abilities is to `extend MouseAdapter`.

Here is a simple example: Each time the mouse is clicked over a frame, the frame's mouse listener prints the position where the mouse was clicked. The class that handles the mouse clicks can be written this simply:

```
import java.awt.event.*;
import javax.swing.*;
public class TestListener extends MouseAdapter
{ public TestListener(JFrame f)
```

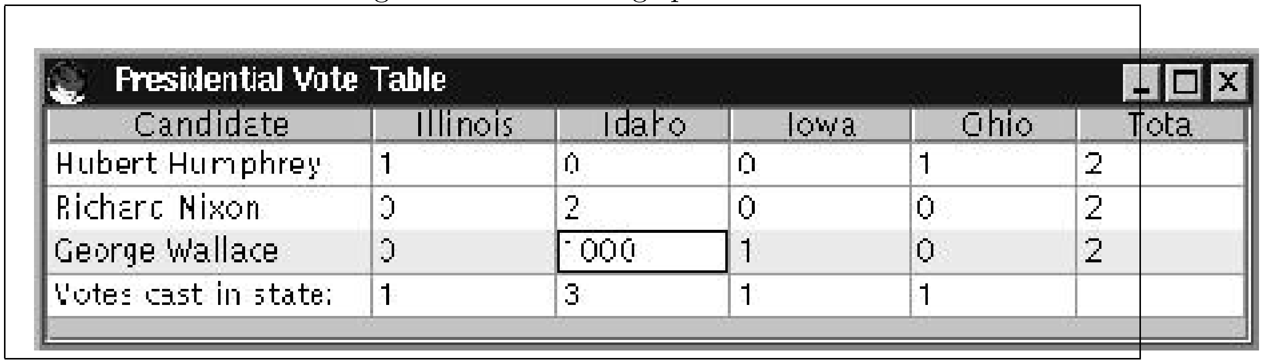
Figure 10.51: view for vote casting

```
import java.awt.*;  import java.awt.event.*;  import javax.swing.*;
/** CastVoteFrame displays a frame that lets a user cast a vote */
public class CastVoteFrame extends JFrame
{ JList state_list;  // the region names from which a user selects
  JList candidate_list;  // the candidates from which a user selects

  /** CastVoteFrame constructs the view
    * @param states - the names of the regions a voter can choose
    * @param candidates - the names of the candidates a voter can choose
    * @param b - the button the user pushes to cast a vote */
  public CastVoteFrame(String[] states, String[] candidates, VoteButton b)
  { Container cp = getContentPane();
    cp.setLayout(new BorderLayout());
      JPanel p1 = new JPanel(new GridLayout(1,2));
      state_list = new JList(states);
      p1.add(state_list);
      candidate_list = new JList(candidates);
      p1.add(candidate_list);
    cp.add(p1, BorderLayout.CENTER);
      JPanel p2  = new JPanel(new FlowLayout());
      p2.add(b);
      b.setViewTo(this);
    cp.add(p2, BorderLayout.SOUTH);
    pack();
    setTitle("Cast a vote");
    setVisible(true);
  }

  /** getInputs returns the inputs the user selected and clears the frame
    * @return  (1) the index of the selected candidate;
    *          (2) the index of the selected state */
  public int[] getInputs()
  { int[] input = new int[2];
    input[0] = candidate_list.getSelectedIndex();
    candidate_list.clearSelection();
    input[1] = state_list.getSelectedIndex();
    state_list.clearSelection();
    return input;
  }
}
```

Figure 10.52: controller and start-up class for voting application

```
import java.awt.*;  import java.awt.event.*;  import javax.swing.*;
/** VoteButton implements a button that computes.... */
public class VoteButton extends JButton implements ActionListener
{ private VoteModel model;  // the table model that is updated
  private CastVoteFrame frame;  // the frame from which the vote is extracted

  public VoteButton(String label, VoteModel m)
  { super(label);
    model = m;
    addActionListener(this);
  }

  public void setViewTo(CastVoteFrame f)
  { frame = f; }

  public void actionPerformed(ActionEvent evt)
  { int[] vote = frame.getInputs();
    if ( vote[0] != -1  &&  vote[1] != -1 )
         { int count = model.getVoteCount(vote[0], vote[1]);
           model.changeVoteCount(vote[0], vote[1], count + 1);
         }
    else { JOptionPane.showMessageDialog(frame,
               "State and Candidate not properly selected---vote ignored");
         }
  }
}

public class Vote2
{ public static void main(String[] args)
  { String[] candidates = {"Hubert Humphrey", "Richard Nixon",
                           "George Wallace"};
    String[] states = {"Illinois", "Idaho", "Iowa", "Ohio"};

    VoteModelWithTotals model = new VoteModelWithTotals(candidates, states);
    VoteFrame view = new VoteFrame(model);
    VoteButton controller = new VoteButton("Vote", model);
    CastVoteFrame frame = new CastVoteFrame(states, candidates, controller);
  }
}
```

Figure 10.53: a voting spreadsheet



Presidential Vote Table

Candidate	Illinois	Idaho	Iowa	Ohio	Tota
Hubert Humphrey	1	0	0	1	2
Richarc Nixon	0	2	0	0	2
George Wallace	0	1000	1	0	2
Votes cast in state:	1	3	1	1	

```
  { super();
    f.addMouseListener(this);  // registers this object as a listener of  f
  }

  public void mouseClicked(MouseEvent e)
  { System.out.println("Mouse clicked at " + e.getX() + ", " + e.getY()); }
}
```

For a frame, v, we state

```
new TestListener(v)
```

and this connects the listener to v.

Mouse listeners work well at helping users draw on panels and frames. For example, we might write an application that translates mouse clicks into tiny red and blue boxes and converts mouse drags into yellow ovals:

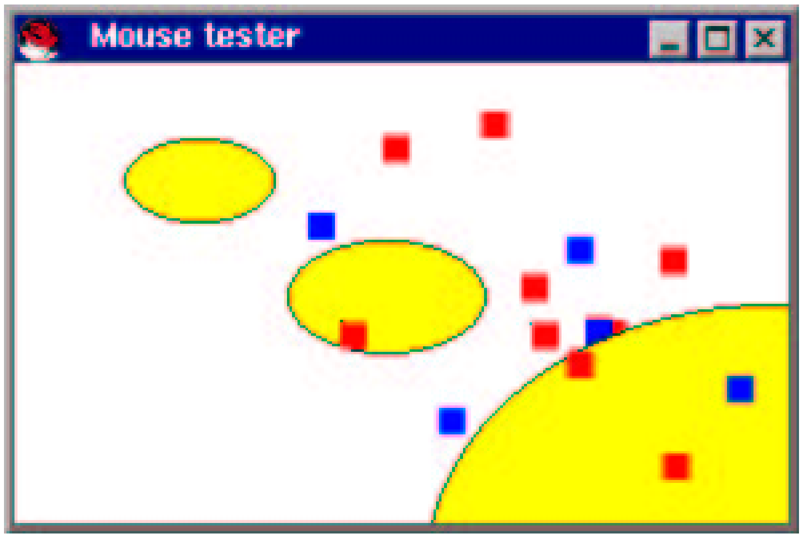


Figure 10.54: table model spreadsheet

```
public class VoteSpreadsheet extends VoteModelWithTotals
{ public VoteSpreadsheet(String[] candidate, String[] region)
  { super(candidate, region); }

  /** isCellEditable  returns whether the cell at  table_row, table_column
    *  can be edited by a user  */
  public boolean isCellEditable(int table_row, int table_column)
  { return ( table_row < (super.getRowCount() - 1)   // not a candidate's total
       &&  table_column > 0                          // not a candidate's name
       &&  table_column < (super.getColumnCount() - 1) ); // not a region total
  }

  /** setValueAt  attempts to insert new_value  into the cell at  table_row,
    *  table_column.  If  new_value  is not an integer, it is ignored */
  public void setValueAt(Object new_value, int table_row, int table_column)
  { String s = (String)new_value;
    try { int i = new Integer(s).intValue();
          changeVoteCount(table_row, table_column-1, i);
        }
    catch(RuntimeException e)
        { } // ignore the user's clumsiness of typing a nonnumeric
  }
}

/** Vote3 starts the spreadsheet application.  */
public class Vote3
{ public static void main(String[] args)
  { String[] candidates = {"Hubert Humphrey", "Richard Nixon",
                           "George Wallace"};
    String[] states = {"Illinois", "Idaho", "Iowa", "Ohio"};
    VoteSpreadsheet model = new VoteSpreadsheet(candidates, states);
    VoteFrame view = new VoteFrame(model);
  }
}
```

Figure 10.55: interface for mouse listeners

```
public interface MouseListener
{ /** mouseClicked handles one or more clicks of a stationary mouse */
  public void mouseClicked(MouseEvent e)

  /** mouseEntered handles the entry of a moving mouse into the component */
  public void mouseEntered(MouseEvent e)

  /** mouseExited handles the exit of a moving mouse from the component */
  public void mouseExited(MouseEvent e)

  /** mousePressed handles the mouse press that initiates a mouse drag */
  public void mousePressed(MouseEvent e)

  /** mouseReleased handles the mouse release that terminates a mouse drag */
  public void mouseReleased(MouseEvent e)
}
```

Figure 10.56: methods of a mouse event

`class MouseEvent`	an event caused by moving a mouse or pressing its button
Methods	
`getClickCount(): int`	Return the number of clicks the user made with the mouse.
`getX(): int`	Return the x-position where the mouse event occurred.
`getY(): int`	Return the y-position where the mouse event occurred.

Figure 10.57: view for drawing shapes on a frame

```
import java.awt.*;  import javax.swing.*;
/** MouseView lets a user draw yellow ovals and tiny colored boxes */
public class MouseView extends JPanel
{ private int width;   // the panel's width
  private int depth;   // the panel's depth

  /** MouseView builds the panel with width w and depth d */
  public MouseView(int w, int d)
  { width = w;  depth = d;
    JFrame my_frame = new JFrame();
    my_frame.getContentPane().add(this);
    my_frame.setSize(width, depth);
    my_frame.setTitle("Mouse tester");
    my_frame.setVisible(true);
  }

  public void paintComponent(Graphics g)
  { g.setColor(Color.white);
    g.fillRect(0, 0, width, depth);
  }

  /** clear erases the frame */
  public void clear()
  { repaint(); }

  /** paintBox paints a small box of color  c  at position  x_pos, y_pos  */
  public void paintBox(int x_pos, int y_pos, Color c)
  { Graphics g = getGraphics();
    g.setColor(c);
    g.fillRect(x_pos, y_pos, 10, 10);
  }

  /** paintOval paints an oval at  x_pos, y_pos  of size width by depth  */
  public void paintOval(int x_pos, int y_pos, int width, int depth)
  { Graphics g = getGraphics();
    g.setColor(Color.yellow);
    g.fillOval(x_pos, y_pos, width, depth);
    g.setColor(Color.black);
    g.drawOval(x_pos, y_pos, width, depth);
  }
}
```

Figure 57 shows a view class that has the methods for painting tiny boxes and yellow ovals. The class's `paintComponent` method clears the frame, meaning that moving or closing/opening the frame makes it lose all previous drawings. If one wishes to retain the drawings, then a model object is required for remembering the shapes and their locations.

Figure 58 shows the controller class that translates the mouse clicks and drags into drawings. The class contains a `main` method that tests the view and controller.

The `mouseClicked` method clears the frame on a double click; a single click draws a box. For fun, the color (red or blue) is chosen based on the location of the click. A mouse drag must be handled in two stages: `mousePressed` saves the position where the drag starts, and `mouseReleased` retrieves this information and calculates the position and size of the oval that the user indicated. Because the user might drag the mouse from right to left and from bottom to top, `Math.min` and `Math.abs` are employed.

Remember that a mouse listener handles mouse events for only that graphical component for which it is registered via `addMouseListener`. If the view object is a frame with buttons, panels, text areas, etc., then each component requires its own mouse listener to process mouse entries, exits, clicks, and presses. And, if a component like a text area is embedded in a frame, then the text area's mouse listener takes precedence over the frame's mouse listener when the mouse enters the text area.

The `MouseListener` interface in Figure 52 reports events related to the mouse's button. Events tied to mouse movement are described by the `MouseMotionListener`, which is documented in the API for `java.awt.event`.

10.15.4 Threads of Execution

How can one application run two animations simultaneously? The throbbing-ball animation in Figure 23 showed that one animation can operate, provided that the controller whose loop runs the animation executes last. (See the last statement of `main` in `class StartThrob`, Figure 25.) But if there are two animations to control, both controllers cannot be "last."

The underlying problem is a Java application has one sequence, or "thread," of instructions to execute. If the thread of instructions leads to a nonterminating loop, the instructions that follow the loop will never be performed. Therefore, to operate two nonterminating loops, two threads of execution are necessary.

The Java language makes it possible to give an object its own thread of execution, so that the object executes as if it were a separate application. More precisely, the object must have a method named `run`, and when the object's own thread of execution is created, the object's `run` method executes.

We can use multiple threads with the throbbing-ball animation so that two balls throb simultaneously. The model class, `ThrobbingBall`, and its view, `ThrobPanel` (Figure 24), are left intact. The first key modification is adding the phrase, `implements Runnable` to the header line of `class ThrobController` in Figure 25:

Figure 10.58: controller for mouse clicks and drags

```
import java.awt.*;
import java.awt.event.*;
/** MouseController controls drawing based on mouse clicks */
public class MouseController extends MouseAdapter
{ private MouseView view;  // the view object where shapes are drawn

  public MouseController(MouseView f)
  { view = f;
    view.addMouseListener(this);
  }

  /** mouseClicked handles a single click by drawing a small box;
    *   it handles a multiple click by clearing the view
    * @param e - the event that remembers the position and number of clicks */
  public void mouseClicked(MouseEvent e)
  { if ( e.getClickCount() > 1 )    // a ``double click''?
         { view.clear(); }          // then, clear the view
    else { int x = e.getX();        // else, draw a box
           int y = e.getY();
           Color c = Color.red;
           if ( (x + y)% 2 == 0 )  // for fun, use blue on even pixel values
              { c = Color.blue; }
           view.paintBox(x, y, c);
         }
  }

  private int start_x;   // remembers the position where the user
  private int start_y;   //   started to drag the mouse

  /** mousePressed remembers the start position of a mouse drag
    * @param e - the event that remembers the position  */
  public void mousePressed(MouseEvent e)
  { start_x = e.getX();
    start_y = e.getY();
  }
 ...
```

Figure 10.58: controller for mouse clicks and drags (concl.)

```
  /** mouseReleased draws an oval based on the end position of the mouse drag
    * @param e - the event that remembers the end position  */
  public void mouseReleased(MouseEvent e)
  { int new_x = Math.min(start_x, e.getX());   // compute upper left corner
    int new_y = Math.min(start_y, e.getY());
    int width = Math.abs(start_x - e.getX());  // compute absolute size
    int depth = Math.abs(start_y - e.getY());
    view.paintOval(new_x, new_y, width, depth);
  }

  /** test the controller with a view:  */
  public static void main(String[] args)
  { new MouseController(new MouseView(300, 200)); }
}
```

```
public class ThrobController implements Runnable
{ ...
  public void run()
  { ... }
  ...
}
```

The interface, `Runnable`, promises that the class contains a method named `run`; this is the method that will execute with its own thread.

Next, we modify the frame that displays the animations so that it shows two panels of throbbing balls:

```
import java.awt.*;
import javax.swing.*;
public class Throb2Frame extends JFrame
{ public Throb2Frame(int size, ThrobPanel p1, ThrobPanel p2)
  { Container cp = getContentPane();
    cp.setLayout(new GridLayout(1,2));  // display the animations side by side
    cp.add(p1);
    cp.add(p2);
    setSize(size * 2 + 50, size + 10);
    setVisible(true);
  }
}
```

The important modifications occur in the start-up class, where each animation controller is given its own thread of execution:

```
/** StartThrobs builds two animation panels and displays them. */
public class StartThrobs
{ public static void main(String[] a)
  { int panel_size = 180;
    ThrobbingBall b1 = new ThrobbingBall();
    ThrobPanel w1 = new ThrobPanel(panel_size, b1);
    ThrobbingBall b2 = new ThrobbingBall();
    ThrobPanel w2 = new ThrobPanel(panel_size, b2);
    new Throb2Frame(200, w1, w2);
    new Thread(new ThrobController(w1, b1, 200)).start();
    new Thread(new ThrobController(w2, b2, 300)).start();
    System.out.println("Animations have started!");
  }
}
```

The phrase, `new Thread(...).start()`, gives an object its own thread of execution for its `run` method, which immediately executes. In the above example, this causes the two panels of animations to run simultaneously. Note also that the `println` statement executes properly in `main`'s thread of execution after the two other threads are created and running.

Threads can be used to build crude simulations where objects appear to have "life" of their own. Consider a simulation of two sales agents, both of whom are selling tickets to an opera. One agent works three times as fast as the other at selling tickets. We might model the sales agents with objects that have their own threads of execution; the objects share an object that holds the opera tickets. Here is the class that models the opera tickets:

```
public class Tickets
{ private int how_many_left;  // quantity of unsold tickets

  public Tickets(int initial_value)
  { how_many_left = initial_value; }

  public boolean ticketsAvailable()
  { return how_many_left > 0; }

  public int sellTicket()
  { int ticket_number = how_many_left;
    how_many_left = how_many_left - 1;
    return ticket_number;
  }
}
```

And here is a sales agent:

```
public class SalesAgent implements Runnable
```

```
{ private int id;                 // the sales agent's ``name''
  private Tickets ticket_source;  // the object that has the tickets
  private int time;               // the time the agent pauses between sales

  public SalesAgent(int i, Tickets t, int speed)
  { id = i;
    ticket_source = t;
    time = speed;
  }

  public void run()
  { int i;
    while ( ticket_source.ticketsAvailable() )
          { i = ticket_source.sellTicket();
            System.out.println("Agent " + id + " sells ticket " + i);
            delay();  // pause so that other threads might resume
          }
    System.out.println("Agent" + id + " is finished");
  }

  private void delay()
  { try{ Thread.sleep(time); }
    catch (InterruptedException e) {}
  }
}
```

The loop in the `run` method sells tickets until no more are available. The `delay` in the loop is crucial, as explained below.

The start-up class creates threads for the two sales agents:

```
public class StartTicketSales
{ public static void main(String[] args)
  { Tickets c = new Tickets(20);
    new Thread(new SalesAgent(1, c, 100)).start();
    new Thread(new SalesAgent(2, c, 300)).start();
  }
}
```

The application executes the two sales agents, which display their sales in the display. As expected, the first sales agent sells three times as many tickets as the second.

The invocation of `delay` within the `SalesAgent`'s `run` method is crucial—on some computers, the computer executes a thread for as long as possible, and only when there is a delay in a thread's execution will the computer resume execution of other threads. Here, the `delay` method ensures that both of the sales agents have opportunities to sell tickets.

Further, there is great danger when distinct threads of execution share an object. To see this, move the invocation, `delay()`, within `run`'s loop to the beginning of the loop:

```
while ( ticket_source.ticketsAvailable() )
      { delay();  // this causes erroneous behavior!
        i = ticket_source.sellTicket();
        System.out.println("Agent " + id + " sells ticket " + i);
      }
```

Because the sales agent pauses between checking ticket availability and selling the ticket, it is possible for two agents to both confirm availability of the *last* available ticket and for both of them to sell it! This is indeed what happens when the above example uses the altered loop.

The above problem is called a *race condition* (both agents are "racing" to sell the last available ticket), and it is one of many problems that can arise when multiple threads share an object. These problems cannot be analyzed further here. As a rule, *do not use the techniques in this section when the order in which a shared object receives messages from multiple threads can affect the outcome of the application.*

10.15.5 GUI Design and Use-Cases

Most of the previous discussion on program design has focussed on model construction. With the inclusion of GUIs, the design of the view becomes equally important to a program's success. We begin by repeating the guidelines for GUI design given at the beginning of this Chapter:

- Organize sequences of events into natural units for processing.
- Make it difficult or impossible for the user to generate a truly nonsensical sequence of events.

To help meet these guidelines, design a GUI so that any order of events generated from a frame can be sensibly processed. If one event simply must occur before another, then make the first event generate a secondary frame or dialog from which the second event can occur. Also, consider all possible orders in which events might occur, and write the controllers so that they respond accordingly. It also helps to design the GUI so that if the user forgets to submit a particular input/event, a default value is automatically supplied. When an input must be one of a finite number of choices (e.g., "pick one: Celsius or Fahrenheit"), a component like a list of two items is better than one like a text field.

The above advice is based on the designer knowing all the events a user might generate—where does the designer get this knowledge? A standard technique for cataloguing events is generating *use-cases.* A "use-case" is a description of the steps that occur when a user submits one request to the application for computation—it is a "case study" of one "use" of the application. By inventing a collection of use-cases, a designer gains insight towards the design of the application's architecture and its GUI.

Here is an example of a use-case for a program that manages the balance in a user's bank account (see Chapter 6): A user deposits $10.50 into her account.

1. The user submits `10.50` to the GUI.

2. The user indicates the amount is a deposit.

3. The user signals that the transaction should be made.

4. Computation occurs, and an acknowledgement, along the with account's new balance, appears on the GUI.

This use-case makes clear that the GUI must have a component that helps the user submit a numerical amount and a component that helps the user select and perform a deposit. There should be a model of the bank account that receives the deposit, and there should be a GUI component that displays the new balance.

The above use-case does not make clear whether text fields, buttons, lists, or labels should be used in the GUI—this is the decision of the designer. The nature of the model is not decided, either. Additional use-cases are needed.

Here is another example of a use-case for the bank-account application:

1. The user submits `10000.00` into the GUI.

2. The user indicates the amount is a withdrawal.

3. The user signals that the transaction should be made.

4. The application reports that the withdrawal is refused because the account's balance is smaller than the withdrawal amount.

This use-case is concerned with an error situation, and it makes clear that the GUI must have a means for reporting the error and helping the user correct the mistake. A third use-case is:

1. Without submitting a numerical amount, the user signals that a withdrawal should be done.

2. The application reports an error.

This use-case reminds the designer that the application should deal with unexpected orders of events and missing input information.

To develop a representative set of use-cases, a designer must discuss the application with its users, study the problem area and its past solutions, and conceive "thought experiments" about what might happen when the application is used.

Based on such use-cases, the application's designer compiles a list of the expected behaviors ("methods") of the application—deposits, withdrawals, account queries, interest calculations, etc.—and the designer constructs a GUI that guides the user through the operation of the application in a way that minimizes the potential for invalid input. (For example, the GUI should make it difficult or impossible for a user to demand a deposit or withdrawal without first specifying an amount.)

Here is a final thought to remember when you design an application's GUI: Users do not like to read instructions. Therefore, an interface should be organized to lead a naive user through a useful execution of a program even when the user is mostly uninformed about how to use the program. One way to help naive users is to limit the possible activities a user might do with a frame. For example, if a frame consists of just one button, the obvious (and only) action that a user could take is to push the button. Of course, real interfaces are rarely so simple, but strive to minimize the number of choices a user makes with any specific window. It is always better to lead a user step-by-step through a program execution with multiple dialogs and frames than it is to design one complex frame that triggers all the program's methods.

If used judiciously, disabling buttons and menu items (through the use of the `setEnabled(false)` method) can effectively limit a user's actions. For example, a text editor's "paste" menu item can be disabled at the times the user has not cut or copied text into the editor's "clipboard." But too much disabling and reenabling buttons can confuse a user as to the state of the application.

10.15.6 Summary of Methods for Graphical Components

The Tables that follow list the methods we used in this Chapter with AWT/Swing components; they complement Figure 5. A more comprehensive listing can be located in the API documentation for the packages `java.awt`, `java.awt.event`, `javax.swing`, `javax.swing.event`, and `javax.table`.

`abstract class Component`	the basic unit for GUI construction in the AWT framework
Methods	
`setSize(int width, int depth)`, `setVisible(boolean yes)`, `setLocation(int x, int y)`, `setBackground(Color c)`, `paint(Graphics g)`, `repaint()`, `getGraphics()`	See Table 18, Chapter 4.
`setForeground(Color c)`	Sets the foreground color of this object to `c` (if such a notion is sensible for the object).
`isShowing(): boolean`	Returns whether the component is showing on the display.
`getLocationOnScreen(): Point`	Returns the coordinates, encased in a `Point` object, of (the upper left corner of) the component's location on the display.
`setLocation(Point p)`	Positions the component to appear at location `p` on the display.
`setFont(Font f)`	Sets the component's text font to `f`.

classes `Point` and `Font`	x,y coordinate positions and text fonts
Constructor	
`new Point(int x, int y)`	Constructs a point representing the pixel position, `x, y`.
Methods	
`translate(int dx, int dy)`	Changes a point's value so that its former position, `x, y`, becomes `x + dx, y + dy`.
Constructor	
`new Font(String fontname, int style, int typesize)`	Creates a new font. `fontname` might be `"Courier"`, `"TimesRoman"`, `"SansSerif"`, and others; `style` might be `Font.PLAIN`, `Font.BOLD`, or `Font.ITALIC`; `typesize` is typically between 8 and 20.

`class Container extends Component`	a component that can hold other components
Methods	
`setLayout(LayoutManager layout)`	Tells the container to use `layout` to arrange the components that are inserted into it.
`add(Component ob)`	Inserts `ob` inside the container at the end of its layout.
`add(Component ob, Object restriction)`	Inserts `ob` inside the container, using the `restriction` to determine `ob`'s position in the layout. (See `class BorderLayout` for examples of restrictions.)

`class Window extends Container`	a top-level component that can be displayed by itself
Methods	
`pack()`	Sets the size of the window just large enough that it displays all the components inserted into it. (Note: Empty panels are not always displayed at full size; use `setSize` in this case.)
`dispose()`	Removes the window from the display and makes it impossible to display it again.
`addWindowListener(WindowListener listener)`	Registers `listener` as a listener (event handler) for window events generated by this object.

`class JFrame extends Window`	a window with a title and menu bar
Methods	
`setTitle(String s)`	See Table 18, Chapter 4.
`getContentPane(): Container`	Returns the frame's content pane, into which components can be inserted.
`setJMenuBar(JMenuBar mbar)`	Attaches menu bar `mbar` to the top of the frame.

`class JApplet extends Container`	a top-level component that can be displayed in a Web page
Methods	
`init()`	Initializes the applet immediately after the applet is constructed.
`getParameter(String s): String`	Fetches from the `applet` command that started the applet the actual parameter labelled by `s`
`getContentPane(): Container`	Returns the applet's content pane, into which components can be inserted.
`setJMenuBar(JMenuBar mbar)`	Attaches menu bar `mbar` to the top of the applet.

`abstract class JComponent extends Container`	the basic graphical object in the Swing framework

`abstract class Abstract Button extends JComponent`	a component that can be "pushed" or "set" or "selected" by a user
Methods	
`addActionListener(ActionListener listener)`	Attaches `listener` to this button as its listener (event handler) for action events.
`setEnabled(boolean b)`	Sets whether or not the button generates an action event when it is pushed.
`isEnabled(): boolean`	Returns whether or not the button generates an action event when it is pushed.
`getText(): String`	Returns the text that appears on the button's face in the view.
`setText(String s)`	Sets the text that appears on the button's face to `s`.
`doClick()`	Generates an action event, just like when the user pushes the button.
`isSelected(): boolean`	Returns whether or not the button is "selected," that is, pushed inwards.
`setSelected(boolean b)`	Sets whether or not the button is "selected," that is, pushed inwards.

`class JButton extends AbstractButton`	a button
Constructor	
`JButton(String s)`	Constructs a button whose face displays `s`.

`class JMenuItem extends AbstractButton`	a menu item
Constructor	
`JMenuItem(String s)`	Constructs a menu item whose face displays `s`.

`class JMenu extends JMenuItem`	a "container" for menu items
Constructor	
`JMenu(String s)`	Constructs a menu whose face displays `s`.
Methods	
`add(Component c)`	Adds component `c` (normally, a menu or menu item) to the end of this menu.
`addSeparator()`	Adds a separator bar to the end of this menu.

`class JLabel extends JComponent`	a component that displays a string
Constructor	
`JLabel(String s)`	Constructs a label that displays `s`.
Methods	
`getText():String`	Returns the text displayed by the label.
`setText(String s)`	Sets the label to display `s`.

`class JList extends JComponent`	a sequence of items that can be selected by the user
Constructor	
`JList(Object[] items)`	Constructs a list whose internal model is the array, `items`. The elements of `items` must be objects that possess a method, `toString(): String`. (Note: `String` objects have this method by default.) The `items` are indexed 0, 1, 2, etc.
Methods	
`setSelectionMode(int mode)`	Sets whether at most one (`ListSelectionModel.SINGLE_SELECTION`) or more than one (`ListSelectionModel.MULTIPLE_INTERVAL_SELECTION`) item can be selected from the list by the user.
`getSelectedIndex(): int`	Returns the index number of the item currently selected. (If no item is selected, -1 is returned. If multiple items are selected, the lowest numbered index is returned.)
`getSelectedIndices(): int[]`	Returns an array whose elements are the index numbers of all items currently selected.
`setSelectedIndex(int i)`	Makes item `i` of the list appear selected in the list's view.
`setSelectedIndices(int[] r)`	Makes all the items in array `r` appear selected in the list's view.
`clearSelection()`	Unselects all selected items.
`addListSelectionListener(ListSelectionListener listener)`	Registers `listener` as a listener (event handler) for list selection events within this object.

`class JMenuBar extends JComponent`	a component to which menus are attached
Constructor	
`JMenuBar()`	Constructs a menu bar.
Methods	
`add(JMenu m)`	Attaches menu `m` to the menu bar.

`class JOptionPane extends JComponent`	displays dialogs
Methods	
`showMessageDialog(Component owner, Object message)`	Displays a message dialog (that is, a dialog that displays `message`, normally a string, and an `OK` button) that suspends the execution of `owner` until the user dismisses the dialog.
`showConfirmDialog(Component owner, Object message): int`	Displays a confirm dialog (that is, a dialog that displays `message`, normally a string, and three buttons, `Yes`, `No`, and `Cancel`). The dialog suspends the execution of `owner` until the user dismisses it. When dismissed, the dialog returns one of these four integer results: `JOptionPane.YES_OPTION`, `JOptionPane.NO_OPTION`, `JOptionPane.CANCEL_OPTION`, or `JOptionPane.CLOSED_OPTION`. (The last occurs when the user pushes the "X"-button to terminate the dialog.)
`showInputDialog(Component owner, Object message): String`	Displays an input dialog (that is, a dialog that displays `message`, a text field, and two buttons, `OK` and `Cancel`). The dialog suspends the execution of `owner` until the user dismisses it. When dismissed by a push of `Ok`, the dialog returns the contents of the text field. Otherwise, the dialog returns `null`.

`class JPanel extends JComponent`	the simplest possible `JComponent`
Constructors	
`JPanel()`	Constructs a panel.
`JPanel(LayoutManager layout)`	Constructs a panel that uses `layout` to arrange its components.

`class JScrollPane extends JComponent`	a scroll bar that attaches to a component
Constructor	
`JScrollPane(Component c)`	Wraps a scroll bar around `c`.

`abstract class JTextComponent extends JComponent`	a component that holds user-editable text
Methods	
`getText(): String`	Returns, as one string, the entire contents of the text component
`setText(String s)`	Resets the contents of the text component to `s`.
`getCaretPosition(): int`	Returns the position of the insertion caret within the text component.
`setCaretPosition(int i)`	Repositions the insertion caret to position `i`, a nonnegative integer.
`moveCaretPosition(int i)`	"Drags" the caret from its exiting position to position `i`, in the process selecting all the text that falls between the old and new positions.
`getSelectedText(): String`	Returns the text that is selected within the text component.
`getSelectionStart(): int`	Returns the position where the selected text begins.
`getSelectionEnd(): int`	Returns the position where the selected text ends.
`replaceSelection(String s)`	Replaces the currently selected text by `s`.
`cut()`	Removes the selected text from the text component and copies it into the component's clipboard.
`copy()`	Copies the selected text into the component's clipboard.
`paste()`	Copies the text in the component's clipboard into the text component at the position marked by the insertion caret.
`isEditable(): boolean`	Returns whether or not the user is allowed to alter the contents of the text component.
`setEditable(boolean b)`	States whether or not the user is allowed to alter the contents of the text component.

`class JTextField extends JTextComponent`	a text component that holds one line of text
Constructor	
`JTextField(String s, int i)`	Constructs a text field that displays `i` columns (that is, can display at most `i` copies of the letter, "`m`") whose initial value is `s`.
Methods	
`addActionListener(ActionListener listener)`	Registers `listener` as a listener (event handler) for action events (that is, presses of *Enter*) generated in this object.

`class JTextArea extends JTextComponent`	a text component that holds multiple lines of text
Constructor	
`JTextArea(String s, int rows, int columns)`	Constructs a text area of size `rows` by `columns` whose initial value is `s`.
Methods	
`setLineWrap(boolean b)`	State whether text lines that are longer than the text area's width should be "wrapped" to appear on the next line.
`insert(String s, int i)`	Insert `s` at position `i` within the text area.
`replaceRange(String s, int start, int end)`	Remove the text starting at position `start` and ending at position `end - 1` and replace it by `s`.

`class Observable`	an object that can generate its own "events" for its Observers
Methods	
`addObserver(Observer ob)`	Registers `ob` as an Observer ("event handler") of this object.
`setChanged()`	Asserts that this object has "changed" its internal state, causing an "event."
`notifyObservers()`	If this object has "changed," (see `setChanged`), then send an `update` message to this object's Observers. Next, reset the object so that it is no longer "changed."
`notifyObservers(Object arg)`	If this object has "changed," (see `setChanged`), then send an `update` message to this object's Observers. Include with the message as its second parameter, `arg`. Next, reset the object so that it is no longer "changed."

`class WindowEvent`	an event generated by the user opening, closing, or adjusting the size of a window
Method	
`getWindow(): Window`	Return the address of the window where this event occurred.

`class ActionEvent`	an event caused by the user clicking a button or selecting a menu item
Methods	
`getSource(): Object`	Return the address of this event's source.
`getActionCommand(): String`	Return the text that appears on the face of the event source object.

`class ListSelectionEvent`	an event caused by the user selecting a list item
Methods	
`getFirstIndex(): int`	Return the index of the first of the items whose selection generated this event.
`getLastIndex(): int`	Return the index of the last of the items whose selection generated this event.