APPENDICES

APPENDIX

A Operators

Figure A.1 shows the precedence and associativity of the common Java operators discussed. The bitwise operators have not been used in this book.

Category	Examples	Associativity
Operations on References	`. []`	Left to right
Unary	`++ -- ! - (type)`	Right to left
Multiplicative	`* / %`	Left to right
Additive	`+ -`	Left to right
Shift (bitwise)	`<< >> >>>`	Left to right
Relational	`< <= > >= instanceof`	Left to right
Equality	`== !=`	Left to right
Boolean (or bitwise) AND	`&`	Left to right
Boolean (or bitwise) XOR	`^`	Left to right
Boolean (or bitwise) OR	`|`	Left to right
Logical AND	`&&`	Left to right
Logical OR	`||`	Left to right
Conditional	`?:`	Right to left
Assignment	`= *= /= %= += -=`	Right to left

Figure A.1 Java operators listed from highest to lowest precedence

APPENDIX

B Graphical User Interfaces

A *graphical user interface* (*GUI*) is the modern alternative to terminal I/O that allows a program to communicate with its user. In a GUI, a window application is created. Some of the ways to perform input include selection from a list of alternatives, pressing buttons, checking boxes, typing in text fields, and using the mouse. Output can be performed by writing into text fields as well as drawing graphics. In Java 1.2 or higher, GUI programming is performed by using the *Swing* package.

In this appendix, we will see:

- The basic GUI components in Swing
- How these components communicate information
- How these components can be arranged in a window
- How to draw graphics

A *graphical user interface* (*GUI*) is the modern alternative to terminal I/O that allows a program to communicate with its user.

B.1 The Abstract Window Toolkit and Swing

The *Abstract Window Toolkit* (*AWT*) is a GUI toolkit that is supplied with all Java systems.

The *Abstract Window Toolkit* (*AWT*) is a GUI toolkit that is supplied with all Java systems. It provides the basic classes to allow user interfaces. These classes can be found in the package `java.awt`.[1] The AWT is designed to be portable and work across multiple platforms. For relatively simple interfaces, the AWT is easy to use. GUIs can be written without resorting to visual development aids, and provides a significant improvement over basic terminal interfaces.

GUI programming is event-driven.

In a program that uses terminal I/O, the program typically prompts the user for input and then executes a statement that reads a line from the terminal. When the line is read, it is processed. The flow of control in this situation is easy to follow. GUI programming is different. In GUI programming, the input components are arranged in a window. After the window is displayed, the program waits for an event, such as a button push, at which point an event handler is called. This means that the flow of control is less obvious in a GUI program. The programmer must supply the event handler to execute some piece of code.

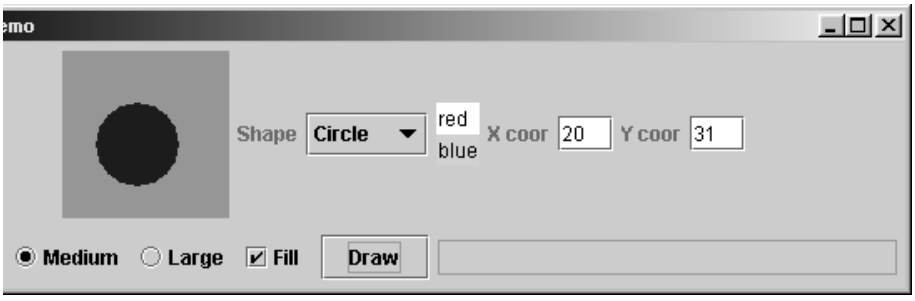

Figure B.1 A GUI that illustrates some of the basic Swing components

1. Code in this appendix uses the wild-card import directive to save space.

Java 1.0 provided an event model that was cumbersome to use. It was replaced in Java 1.1 by a more robust event model. Not surprisingly, these models are not entirely compatible. Specifically, a Java 1.0 compiler will not successfully compile code that uses the new event model. Java 1.1 compilers will give diagnostics about Java 1.0 constructs. However, already compiled Java 1.0 code can be run by a Java 1.1 interpreter. This appendix describes the newer event model only. Many of the classes required by the new event model are found in the `java.awt.event` package.

The event model changed in incompatible ways from Java 1.0 to Java 1.1. The latter version is described here.

The AWT provided a simple GUI, but was criticized for its lack of flair, as well as poor performance. In Java 1.2, an improved set of components was added in a new package called `javax.swing`. These components are known as *Swing*. Components in Swing look much better than their AWT counterparts, there are new Swing components that did not exist in AWT (such as sliders and progress bars), and have many more options (such as easy tooltips and mnemonics). Additionally, Swing provides the notion of look-and-feel, in which a programmer can display the GUI in Windows, X-Motif, Macintosh, platform independent (metal), or even customized style, regardless of the underlying platform (although, because of copyright issues and perhaps bad blood between Sun and Microsoft, Windows look-and-feel works only on Windows systems).

Swing is a GUI package provided in Java 1.2 that is built on top of the AWT and provides slicker components.

Swing is built on top of the AWT, and as a result, the event-handling model is unchanged. Programming in Swing is very similar to the programming in Java 1.1 AWT, except that many names have changed. In this appendix we describe Swing programming only. Swing is a large library; it is not unusual to see entire

books devoted to the topic, so our presentation greatly understates the issues that are involved in user interface design.

Figure B.1 illustrates some of the basic components provided by Swing. These include the `JComboBox` (currently *Circle* is selected), a `JList` (currently *blue* is selected), basic `JTextField`s for input, four `JCheckBox`es, and a `JButton` (named *Draw*). Next to the button is a `JTextField` that is used for output only (hence, it is darker than the input `JTextField`s above it). In the top left-hand corner is a `JPanel` object that can be used for drawing pictures and handling mouse input.

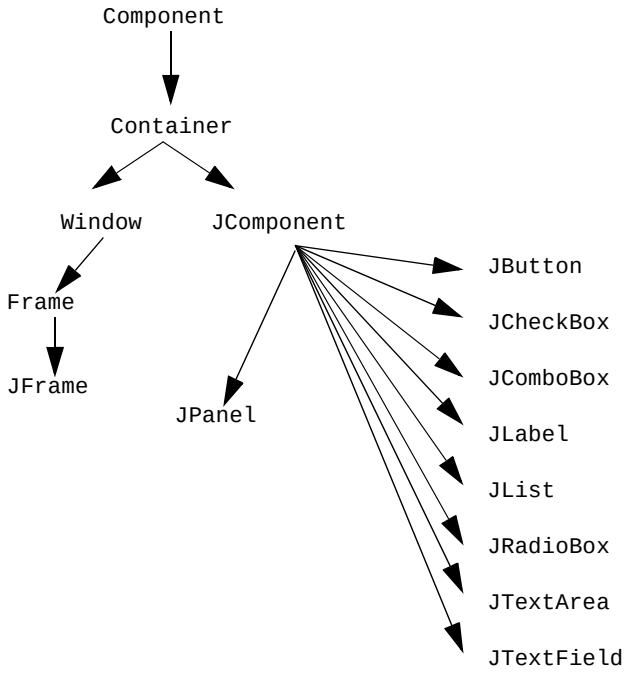

Figure B.2 Compressed hierarchy of Swing

This appendix describes the basic organization of the Swing API. It covers the different types of objects, how they can be used to perform input and output, how these objects are arranged in a window, and how events are handled.

B.2 Basic Objects in the AWT

The AWT is organized using a class inheritance hierarchy. A compressed version of this hierarchy is shown in Figure B.2. This is compressed because some intermediate classes are not shown. For instance, in the full hierarchy, `JTextField` and `JTextArea` are extended from `JTextComponent`, while many classes that deal with fonts, colors, and other objects and are not in the `Component` hierarchy are not shown at all. The classes `Font` and `Color`, which are defined in the `java.awt` package, are extended from `Object`.

B.2.1 Component

The *Component* class is an abstract class that is the superclass of many AWT objects. It represents something that has a position and a size and can be painted on the screen as well as can receive input events.

The *Component* class is an abstract class that is the superclass of many AWT objects, and thus Swing objects. Because it is abstract, it cannot be instantiated. A `Component` represents something that has a position and a size and can be painted on the screen as well as can receive input events. Some examples of the `Component` are evident from Figure B.2.

The `Component` class contains many methods. Some of these can be used to specify the color or font; others are used to handle events. Some of the important methods are

```
void setSize( int width, int height );
void setBackground( Color c );
void setFont( Font f );
void show( );
```

The `setSize` method is used to change the size of an object. It works with `JFrame` objects, but it should not be called for objects that use an automatic layout, such as `JButton`s. For those, use `setPreferredSize`; this method takes a `Dimension` object that itself is constructed with a length and width (and is defined in `JComponent`). The `setBackground` and `setFont` methods are used to change the background color and font associated with a `Component`. They require a `Color` and `Font` object, respectively. Finally, the `show` method makes a component visible. Its typical use is for a `JFrame`.

B.2.2 Container

A *Container* is the abstract superclass representing all components that can hold other components.

In the AWT, a *Container* is the abstract superclass representing all components that can hold other components. An example of an AWT `Container` is the `Window` class, which represents a top-level window. As the inheritance hierarchy shows, a `Container` IS-A `Component`. A particular instance of a `Container` object will store a collection of `Component`s as well as other `Container`s.

The container has a useful helper object called a `LayoutManager`, which is a class that positions components inside the container. Some useful methods are

```
void setLayout( LayoutManager mgr );
void add( Component comp );
void add( Component comp, Object where );
```

Layout managers are described in Section B.3.1. A container must first define how objects in the container should be arranged. This is done by using `setLayout`. It then adds the objects into the container one-by-one by using `add`. Think of the container as a suitcase, in which you can add clothes. Think of the layout manager as the packing expert who will explain how clothes are to be added to the suitcase.

B.2.3 Top-level Containers

The basic containers are the top-level *Window* and *JComponent*. The typical heavyweight components are JWindow, JFrame, and JDialog.

As Figure B.2 shows, there are two types of `Container` objects, namely

1. the top-level windows which eventually reaches `JFrame`
2. the `JComponent`, which eventually reaches most other Swing components.

`JFrame` is an example of a "heavyweight component," while all Swing components in the `JComponent` hierarchy are "lightweight." The basic difference between heavyweight and lightweight components is that lightweight components are drawn on a canvas entirely by Swing whereas heavyweight components interact with the native windowing system. As a result, whereas lightweight components can add other lightweight components, (for instance, you can use `add` to place several `JButton` objects in a `JPanel`), but you should not `add` directly into a heavyweight component. Instead you obtain a `Container` representing its "content pane" and `add` into the content pane, thus allowing Swing to update the content pane. Thus the native windowing system is not involved in the update (you will get a runtime exception if you attempt to add into a heavyweight component), increasing update performance.

There are only a few basic top-level windows, including:

1. `JWindow`: a top-level window that has no border
2. `JFrame`: a top-level window that has a border and can also have an associated `JMenuBar`[2]
3. `JDialog`: a top-level window used to create dialogs

An application that uses a Swing interface should have a `JFrame` (or a class extended from `JFrame`) as the outermost container.

2. Menus are not discussed in this Appendix.

B.2.4 JPanel

The other Container subclass is the JComponent. One such JComponent is the *JPanel* which is used to store a collection of objects, but does not create borders: So it is the simplest of the container classes.

The *JPanel* is used to store a collection of objects, but does not create borders. As such, it is the simplest of the Container classes.

The primary use of the JPanel is to organize objects into a unit. For instance, consider a registration form that requires a name, address, social security number, and home and work telephone numbers. All of these form components might produce a PersonPanel. Then the registration form could contain several PersonPanel entities to allow the possibility of multiple registrants.

As an example, Figure B.3 shows how the components shown in Figure B.1 are grouped into a JPanel class and illustrates the general technique of creating a subclass of JPanel. It remains to construct the objects, lay them out nicely, and handle the button push event.

Note that GUI implements the ActionListener interface. This means that it understands how to handle an *action event* (in this case, a button push). To implement the ActionListener interface, a class must provide an actionPerformed method. Also, when the button generates an action event, it must know which component is to receive the event. In this case, by making the call at 11 (in Figure B.3), the GUI object that contains the JButton tells the Button to send it the event. These event-handling details are discussed in Section B.3.3.

A second use of the JPanel is the grouping of objects into a unit for the purpose of simplifying layouts. This is discussed in Section B.3.5.

Almost all of the `JPanel` functionality is in fact inherited from `JComponent`. This includes routines for painting, sizing, and event handling and the method to set tooltips:

```
void setToolTipText( String txt );
void setPreferredSize( Dimension d );
```

B.2.5 Important I/O Components

Swing provides a set of components that can be used to perform input and output. These components are easy to set up and use. The code in Figure B.4 (page 941) illustrates how each of the basic components that are shown in Figure B.1 are constructed. Generally, this involves calling a constructor and applying a method to customize a component. This code does not specify how items are arranged in the `JPanel` or how the states of the components are examined. Recall that GUI programming consists of drawing the interface and then waiting for events to occur. Component layout and event handling is discussed in Section B.3.

`JLabel`

A *`JLabel`* is a component for placing text in a container. Its primary use is to label other components.

A *`JLabel`* is a component for placing text in a container. Its primary use is to label other components such as a `JComboBox`, `JList`, `JTextField`, or `JPanel` (many other components already have their names displayed in some way). In Figure B.1, the phrases *Shape*, *X Coor*, and *Y Coor* are labels. A `JLabel` is constructed with an optional `String` and can be changed with the method `setText`. These methods are

```
JLabel( );
JLabel( String theLabel );
void setText( String theLabel );
```

```
 1 import java.awt.*;
 2 import java.awt.event.*;
 3 import java.awt.swing.*;
 4
 5 class GUI extends JPanel implements ActionListener
 6 {
 7     public GUI( )
 8     {
 9         makeTheObjects( );
10         doTheLayout( );
11         theDrawButton.addActionListener( this );
12     }
13         // Make all the objects
14     private void makeTheObjects( )
15       { /* Implementation in Figure B.4 */ }
16
17         // Layout all the objects
18     private void doTheLayout( )
19       { /* Implementation in Figure B.7 */ }
20
21         // Handle the draw button push
22     public void actionPerformed( ActionEvent evt )
23       { /* Implementation in Figure B.9 */ }
24
25     private GUICanvas    theCanvas;
26     private JComboBox    theShape;
27     private JList        theColor;
28     private JTextField   theXCoor;
29     private JTextField   theYCoor;
30     private JRadioButton smallPic;
31     private JRadioButton mediumPic;
32     private JRadioButton largePic;
33     private JCheckBox    theFillBox;
34     private JButton      theDrawButton;
35     private JTextField   theMessage;
36 }
```

Figure B.3 Basic GUI class shown in Figure B.1

JButton

The *JButton* is used to create a labeled button. When it is pushed, an *action event* is generated.

The *JButton* is used to create a labeled button. Figure B.1 contains a `JButton` with the label *Draw*. When the `JButton` is pushed, an *action event* is generated. Section B.3.3 describes how action events are handled. The `JButton` interface is similar to the `JLabel`. Specifically, a `JButton` is constructed with an optional `String`. The `JButton` label can be changed with the method `setText`. These methods are

```
1         // Make all the objects
2     private void makeTheObjects( )
3     {
4         theCanvas = new GUICanvas( );
5         theCanvas.setBackground( Color.green );
6         theCanvas.setPreferredSize( new Dimension( 99, 99 ) );
7
8         theShape = new JComboBox( new String[]
9                                   { "Circle", "Square" } );
10
11        theColor = new JList( new String[] { "red", "blue" } );
12        theColor.setSelectionMode(
13                           ListSelectionModel.SINGLE_SELECTION );
14        theColor.setSelectedIndex( 0 ); // make red default
15
16        theXCoor = new JTextField( 3 );
17        theYCoor = new JTextField( 3 );
18
19        ButtonGroup theSize = new ButtonGroup( );
20        smallPic = new JRadioButton( "Small", false );
21        mediumPic = new JRadioButton( "Medium", true );
22        largePic = new JRadioButton( "Large", false );
23        theSize.add( smallPic );
24        theSize.add( mediumPic );
25        theSize.add( largePic );
26
27        theFillBox = new JCheckBox( "Fill" );
28        theFillBox.setSelected( false );
29
30        theDrawButton = new JButton( "Draw" );
31
32        theMessage = new JTextField( 25 );
33        theMessage.setEditable( false );
34    }
```

Figure B.4 Code that constructs the objects in Figure B.1

```
JButton( );
JButton( String theLabel );
void setText( String theLabel );
void setMnemonic( char c );
```

JComboBox

The *JComboBox* is used to select a single object (typically a string) via a pop-up list of choices. Only one choice can be selected at any time, and by default only an object that is one of the choices can be selected. If the JComboBox is made

The *JComboBox* is used to select a single string via a pop-up list of choices.

editable, the user can type in an entry that is not one of the choices. In Figure B.1, the type of shape is a `JComboBox` object; *Circle* is currently selected. Some of the `JComboBox` methods are

```
JComboBox( );
JComboBox( Object[] choices );
void   addItem( Object item );
Object getSelectedItem( );
int    getSelectedIndex( );
void   setEditable( boolean edit );
void   setSelectedIndex( int index );
```

A `JComboxBox` is constructed with no parameters or with an array of options. `Object`s (typically strings) can then be added to (or removed from) the list of `JComboxBox` options. When `getSelectedItem` is called, an `Object` representing the current selected item (or `null` if no choice is selected) is returned. Instead of returning the actual `Object`, its index (as computed by the order of calls to `addItem`) can be returned by calling `getSelectedIndex`. The first item added has index 0, and so on. This can be useful because if an array stores information corresponding to each of the choices, `getSelectedIndex` can be used to index this array. The `setSelectedIndex` method is used to specify a default selection.

JList

The *JList* component allows the selection from a scrolling list of Objects. In Figure B.1, the choice of colors is presented as a JList. The JList differs from the JComboBox in three fundamental ways:

The *List* component allows the selection from a scrolling list of Objects. It can be set up to allow for either one selected item or multiple selected items.

1. The JList can be set up to allow either one selected item or multiple selected items (the default is multiple selection).
2. The JList allows the user to see more than one choice at a time.
3. The JList will take up more screen real estate than the Choice.

The basic JList methods are

```
JList( );
JList( Object [] items );
void       setListData( Object [] items );
int        getSelectedIndex( );
int [ ]    getSelectedIndices( );
Object     getSelectedValue( );
Object [ ] getSelectedValues( );
void       setSelectedIndex( int index );
void       setSelectedValue( Object value );
void       setSelectionMode( int mode );
```

A JList is constructed with either no parameters or an array of items (there are other constructors that are more sophisticated). Most of the listed methods have the same behavior (with possibly different names) as the corresponding methods in JComboxBox. getSelectedValue returns null if no items are selected. getSelectedValues is used to handle multiple selection; it returns an array of Objects (possibly length 0) corresponding to the selected items. As with the JComboxBox, indices instead of Objects can be obtained by other public methods.

`setSelectionMode` is used to allow only single item selection. The boilerplate code is:

```
lst.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
```

`JCheckBox` *and* `JRadioButton`

A *`JCheckBox`* is a GUI component that has an *on* state and an *off* state. The *on* state is `true` and the *off* state is `false`. It is considered a button (a class `AbstractButton` is defined in the Swing API from which `JButton`, `JCheckBox`, and `JRadioButton` are all derived). A `JRadioButton` is similar to a `JCheckBox`, except that `JRadioButton`s are round. Figure B.1 contains four `JCheckBox` objects. In this figure, the *Fill* check box is currently `true` and the three other check boxes are in a *`ButtonGroup`*: Only one `JCheckBox` in the group of three may be true. When a `JCheckBox` in a group is selected, all the others in the group are deselected. A `ButtonGroup` is constructed with zero parameters. Note that it is not a `Component`; it is simply a helper class that extends `Object`.

A *`JCheckBox`* is a GUI component that has an *on* state and an *off* state. A *`ButtonGroup`* can contain a set of `buttons` in which only one may be `true` at a time.

The common methods for `JCheckBox` are similar to `JRadioButton` and are:

```
JCheckBox( );
JCheckBox( String theLabel );
JCheckBox( String theLabel, boolean state );
boolean isSelected( );
void    setLabel( );
void    setSelected( boolean state );
```

A stand-alone `JCheckBox` is constructed with an optional label. If a label is not provided, it can be added later with `setLabel`. `setLabel` can also be used to change the existing `JCheckBox` label. `setSelected` is most commonly used to set a default for a standalone `JCheckBox`. `isSelected` returns the state of a `JCheckBox`.

A `JCheckBox` that is part of a `ButtonGroup` is constructed as usual and is then added to the `ButtonGroup` object by use of the `ButtonGroup add` method. The `ButtonGroup` methods are:

```
ButtonGroup( );
void add( AbstractButton b );
```

Canvasses

A canvas component represents a blank rectangular area of the screen onto which the application can draw or receive input events.

In the AWT, a *`Canvas`* component represents a blank rectangular area of the screen onto which the application can draw. Primitive graphics are described in Section B.3.2. A `Canvas` could also receive input from the user in the form of mouse and keyboard events. The `Canvas` was never used directly: Instead, the programmer defined a subclass of `Canvas` with appropriate functionality. The subclass overrode the method

```
void paint( Graphics g );
```

In Swing, this is no longer in vogue. The same effect is obtained by extending `JPanel` and overriding the method

```
void paintComponent( Graphics g );
```

Although this works for any component, by using a `JPanel` of a preferred size, one can avoid having any painting run over the boundary of the "canvassing area."

`JTextField` *and* `JTextArea`*s*

A *`JTextField`* is a component that presents the user with a single line of text. A *`JTextArea`* allows multiple lines and has similar functionality. Thus only `JTextField` is considered here. By default, the text can be edited by the user, but it is possible to make the text uneditable. In Figure B.1, there are three `JTextField` objects: two for the coordinates and one, which is not editable by the user, that is used to communicate error messages. The background color of an uneditable text field differs from that of an editable text field. Some of the common methods associated with `JTextField` are

A *`JTextField`* is a component that presents the user with a single line of text. A *`JTextArea`* allows multiple lines and has similar functionality.

```
JTextField( );
JTextField( int cols );
JTextField( String text, int cols );
String getText( );
boolean isEditable( );
void   setEditable( boolean editable );
void   setText( String text );
```

A `JTextField` is constructed either with no parameters or by specifying an initial optional text and the number of columns. The `setEditable` method can be used to disallow input into the `JTextField`. `setText` can be used to print messages into the `JTextField`, and `getText` can be used to read from the `JTextField`.

B.3 Basic Principles

This section examines three important facets of AWT programming. First, how objects are arranged inside a container, followed by how events, such as button pushing, are handled. Finally, it describes how graphics are drawn inside `Canvas` objects.

B.3.1 Layout Managers

A layout manager automatically arranges components of the container. It is associated with a container by issuing the `setLayout` command. An example of using `setLayout` is the call

The *layout manager* automatically arranges components of the container. A layout manager is associated with a container by the `setLayout` method.

```
setLayout( new FlowLayout( ) );
```

Notice that a reference to the layout manager need not be saved. The container in which the `setLayout` command is applied stores it as a private data member. When a layout manager is used, requests to resize many of the components, such as buttons, do not work because the layout manager will choose its own sizes for the components, as it deems appropriate. The idea is that the layout manager will determine the best sizes that allow the layout to meet the specifications.

Think of the layout manager as an expert packer hired by the container to make the final decisions about how to pack items that are added to the container.

FlowLayout

The simplest of the layouts is the *FlowLayout*, which adds components in a row from left to right.

The simplest of the layouts is the *FlowLayout*. When a container is arranged using the `FlowLayout`, its components are added in a row from left to right. When there is no room left in a row, a new row is formed. By default, each row is centered. This can be changed by providing an additional parameter in the constructor with the value `FlowLayout.LEFT` or `FlowLayout.RIGHT`.

The problem with using a `FlowLayout` is that a row may break in an awkward place. For instance, if a row is too short, a break may occur between a `JLabel` and a `JTextField`, even though logically they should always remain adjacent. One way to avoid this is to create a separate `JPanel` with those two elements and then add the `JPanel` into the container. Another problem with the `FlowLayout` is that it is difficult to line up things vertically.

The `FlowLayout` is the default for a `JPanel`.

Figure B.5 Five buttons arranged using `BorderLayout`

```
 1 import java.awt.*;
 2 import javax.swing.*;
 3
 4     // Generate Figure B.5
 5 public class BorderTest extends JFrame
 6 {
 7     public static void main( String [ ] args )
 8     {
 9         JFrame f = new BorderTest( );
10         JPanel p = new JPanel( );
11
12         p.setLayout( new BorderLayout( ) );
13         p.add( new JButton( "North" ), "North" );
14         p.add( new JButton( "East" ), "East" );
15         p.add( new JButton( "South" ), "South" );
16         p.add( new JButton( "West" ), "West" );
17         p.add( new JButton( "Center" ), "Center" );
18
19         Container c = f.getContentPane( );
20         c.add( p );
21         f.pack( );   // Resize frame to minimum size
22         f.show( );   // Display the frame
23     }
24 }
```

Figure B.6 Code that illustrates `BorderLayout`

`BorderLayout`

`BorderLayout` is the default for objects in the `Window` hierarchy, such as `JFrame`. It lays out a container by placing components in one of five locations. For this to happen, the `add` method must provide as a second parameter one of the strings `"North"`, `"South"`, `"East"`, `"West"`, and `"Center"`; the second parameter defaults to `"Center"` if not provided (so one single-parameter `add` will work, but several `add`s place items on top of each other). Figure B.5 shows five buttons added to a `Frame` using a `BorderLayout`. The code to generate this layout is shown in Figure B.6. Observe that we use the typical idiom of adding into a lightweight `JPanel`, and then adding the `JPanel` into the top-level `JFrame`'s content pane. Typically, some of the five locations may be

`BorderLayout` is the default for objects in the `Window` hierarchy, such as `JFrame` and `JDialog`. It lays out a container by placing components in one of five locations.

unused. Also, the component placed in a location is typically a `JPanel` that contains other components using some other layout.

As an example, the code in Figure B.7 (page 952) shows how the objects in Figure B.1 are arranged. Here, we have two rows, but we want to ensure that the checkboxes, buttons, and output text field are placed below the rest of the GUI. The idea is to create a `JPanel` that stores the items that should be in the top half and another `JPanel` that stores the items in the bottom half. These two `JPanel`s can be placed on top of each other by arranging them using a `BorderLayout`.

Lines 4 and 5 create the two `JPanel` objects `topHalf` and `bottomHalf`. Each of the `JPanel` objects are then separately arranged using a `FlowLayout`. Notice that the `setLayout` and `add` methods are applied to the appropriate `JPanel`. Because the `JPanel`s are arranged with the `FlowLayout`, they may consume more than one row if there is not enough horizontal real estate available. This could cause a bad break between a `JLabel` and a `JTextField`. It is left as an exercise for the reader to create additional `JPanel`s to ensure that any breaks do not disconnect a `JLabel` and the component it labels. Once the `JPanel`s are done, we use a `BorderLayout` to line them up. This is done at lines 28 to 30. Notice also that the contents of both `JPanel`s are centered. This is a result of the `FlowLayout`. To have the contents of the `JPanel`s left-aligned, lines 8 and 19 would construct the `FlowLayout` with the additional parameter `FlowLayout.LEFT`.

When the `BorderLayout` is used, any `add` commands that are issued without a `String` use `"Center"` as the default. If a `String` is provided, but is not one of the acceptable five (including having correct case), then a runtime exception is thrown.[3]

When the `BorderLayout` is used, an `add` command that is issued without a `String` defaults to `"Center"`.

`null` *Layout*

The *`null` layout* is used to perform precise positioning. In the `null` layout, each object is added to the container by `add`. Its position and size may then be set by calling the `setBounds` method:

The *`null` layout* is used to perform precise positioning.

```
void setBounds( int x, int y, int width, int height )
```

Here `x` and `y` represent the location of the upper left-hand corner of the object, relative to the upper left-hand corner of its container. `width` and `height` represent the size of the object. All units are pixels.

The `null` layout is platform-dependent; typically, this is a large liability.

Fancier Layouts

Java also provides the `CardLayout`, `GridLayout`, and `GridBagLayout`. The `CardLayout` simulates the tabbed index cards popular in Windows applications but looks terrible in the AWT. The `GridLayout` adds components into a grid but will make each grid entry the same size. This means that components are stretched in sometimes unnatural ways. It is useful for when this is not a problem,

Other layouts simulate tabbed index cards and allow arranging over an arbitrary grid.

3. Note that in Java 1.0, the arguments to `add` were reversed and missing or incorrect `String`s were quietly ignored, thus leading to difficult debugging. The old style is still allowed, but it is officially discouraged.

such as a calculator keypad that consists of a two-dimensional grid of buttons. The `GridBagLayout` adds components into a grid but allows components to cover several grid cells. It is more complicated than the other layouts.

```
1         // Layout all the objects
2     private void doTheLayout( )
3     {
4         JPanel topHalf    = new JPanel( );
5         JPanel bottomHalf = new JPanel( );
6
7             // Layout the top half
8         topHalf.setLayout( new FlowLayout( ) );
9         topHalf.add( theCanvas );
10        topHalf.add( new JLabel( "Shape" ) );
11        topHalf.add( theShape );
12        topHalf.add( theColor );
13        topHalf.add( new JLabel( "X coor" ) );
14        topHalf.add( theXCoor );
15        topHalf.add( new JLabel( "Y coor" ) );
16        topHalf.add( theYCoor );
17
18            // Layout the bottom half
19        bottomHalf.setLayout( new FlowLayout( ) );
20        bottomHalf.add( smallPic );
21        bottomHalf.add( mediumPic );
22        bottomHalf.add( largePic );
23        bottomHalf.add( theFillBox );
24        bottomHalf.add( theDrawButton );
25        bottomHalf.add( theMessage );
26
27            // Now layout GUI
28        setLayout( new BorderLayout( ) );
29        add( topHalf, "North" );
30        add( bottomHalf, "South" );
31    }
```

Figure B.7 Code that lays out the objects in Figure B.1

Visual Tools

Commercial products include tools that allow the programmer to draw the layout using a CAD-like system. The tool then produces the Java code to construct the objects and provide a layout. Typically, it generates an arrangement using a `null`

layout manager. Even with this system, the programmer must still write most of the code, including the handling of events, but is relieved of the dirty work involved in calculating precise object positions.

B.3.2 Graphics

As mentioned in Section B.2.5, graphics are drawn by using a `JPanel` object. Specifically, to generate graphics, the programmer must define a new class that extends `JPanel`. This new class provides a constructor (if a default is unacceptable), overrides a method named `paintComponent`, and provides a public method that can be called from the canvas's container. The `paintComponent` method is

Graphics are drawn by defining a class that extends `JPanel`. The new class overrides the `paintComponent` method and provides a public method that can be called from the canvas's container.

```
void paintComponent( Graphics g );
```

`Graphics` is an abstract class that defines several methods. Some of these are

`Graphics` is an abstract class that defines several drawing methods.

```
void drawOval( int x, int y, int width, int height );
void drawRect( int x, int y, int width, int height );
void fillOval( int x, int y, int width, int height );
void fillRect( int x, int y, int width, int height );
void drawLine( int x1, int x2, int y1, int y2 );
void drawString( String str, int x, int y );
void setColor( Color c );
```

In Java, coordinates are measured relative to the upper left-hand corner of the component. `drawOval`, `drawRect`, `fillOval`, and `fillRect` all draw an object of specified `width` and `height` with the upper left-hand corner at coordinates given by `x` and `y`. `drawLine` and `drawString` draw lines and text,

In Java, coordinates are measured relative to the upper left-hand corner of the component.

respectively. `setColor` is used to change the current color; the new color is used by all drawing routines until it is changed.

It is important that the first line of `paintComponent` calls the superclass' `paintComponent`.

Figure B.8 (page 955) illustrates how the canvas in Figure B.1 is implemented. The new class `GUICanvas` extends `JPanel`. It provides various private data members that describe the current state of the canvas. The default `GUICanvas` constructor is reasonable, so we accept it.

The data members are set by the public method `setParams`, which is provided so that the container (that is, the `GUI` class that stores the `GUICanvas`) can communicate the state of its various input components to the `GUICanvas`. `setParams` is shown at lines 3 to 13. The last line of `setParams` calls the method `repaint`.

The `repaint` method schedules a component clearing and subsequent call to `paintComponent`. Thus all we need to do is to write a `paintComponent` method that draws the canvas as specified in the class data members. As can be seen by its implementation in lines 15 to 35, after chaining up to the superclass, `paintComponent` simply calls the `Graphics` methods described previously in this appendix.

```
class GUICanvas extends JPanel
{
    public void setParams( String aShape, String aColor, int x,
                           int y, int size, boolean fill )
    {
        this.theShape = aShape;
        this.theColor = aColor;
        xcoor = x;
        ycoor = y;
        theSize = size;
        fillOn = fill;
        repaint( );
    }

    public void paintComponent( Graphics g )
    {
        super.paintComponent( g );
        if( theColor.equals( "red" ) )
            g.setColor( Color.red );
        else if( theColor.equals( "blue" ) )
            g.setColor( Color.blue );

        theWidth = 25 * ( theSize + 1 );

        if( theShape.equals( "Square" ) )
            if( fillOn )
                g.fillRect( xcoor, ycoor, theWidth, theWidth );
            else
                g.drawRect( xcoor, ycoor, theWidth, theWidth );
        else if( theShape.equals( "Circle" ) )
            if( fillOn )
                g.fillOval( xcoor, ycoor, theWidth, theWidth );
            else
                g.drawOval( xcoor, ycoor, theWidth, theWidth );
    }

    private String theShape = "";
    private String theColor = "";
    private int xcoor;
    private int ycoor;
    private int theSize;  // 0 = small, 1 = med, 2 = large
    private boolean fillOn;
    private int theWidth;
}
```

Figure B.8 Basic canvas shown in top left-hand corner of Figure B.1

B.3.3 Events

Java's original event-handling system was cumbersome and has been completely redone.

When the user uses the mouse or types on the keyboard, the operating system produces an event. Java's original event-handling system was cumbersome and has been completely redone. The new model, in place since Java 1.1, is much simpler to program than the old. Note that the two models are incompatible: Java 1.1 events are not understood by Java 1.0 compilers and vice versa. The basic rules are as follows:

1. Any class that is willing to provide code to handle an event must `implement` a *listener* interface. Examples of listener interfaces are `ActionListener`, `WindowListener`, and `MouseListener`. As usual, implementing an interface means that all methods of the interface must be defined by the class.
2. An object that is willing to handle the event generated by a component must register its willingness with an *add listener* message sent to the event-generating component. When a component generates an event, the event will be sent to the object that has registered to receive it. If no object has registered to receive it, then it is ignored.

An action event is generated when the user presses a `JButton`; it is handled by an `actionListener`.

For an example, consider the action event, which is generated when the user presses a `JButton`, hits *Return* while in a `JTextField`, or selects from a `JList` or `JMenuItem`. The simplest way to handle the `JButton` click is to have its container implement `ActionListener` by providing an `actionPerformed` method and registering itself with the `JButton` as its event handler.

This is shown for our running example in Figure B.1 as follows. Recall that in Figure B.3, we already have done two things. At line 5, `GUI` declares that it implements the `ActionListener`, and at line 11, an instance of `GUI` registers itself as its `JButton`'s action event handler. In Figure B.9 (page 959), we imple-

ment the listener by having `actionPerformed` call `setParam` in the `GUICanvas` class. This example is simplified by the fact that there is only one `JButton`, so when `actionPerformed` is called, we know what to do. If `GUI` contained several `JButton`s and it registered to receive events from all of these `JButton`s, then `actionPerformed` would have to examine the `evt` parameter to determine which `JButton` event was to be processed: This would probably involve a sequence of `if`/`else` tests.[4] The `evt` parameter, which in this case is an `ActionEvent` reference, is always passed to an event handler. The event will be specific to the type of handler (`ActionEvent`, `WindowEvent`, and so on), but it will always be a subclass of `AWTEvent`.

A window closing event is generated when an application is closed.

An important event that needs to be processed is the window closing event. This event is generated when an application is closed by pressing on the ☒ that is at the top right-hand corner of the application window. Unfortunately, by default, this event is ignored, so if an event handler is not provided, the normal mechanism for closing an application will not work.

4. One way to do this is to use `evt.getSource( )`, which returns a reference to the object that generated the event.

The window-closing event is handled by implementing the `WindowListener` interface.

Window closing is one of several events that is associated with a `WindowListener` interface. Because implementing the interface requires us to provide implementations for many methods (which are likely to be empty bodies), the most reasonable course of action is to define a class that extends `JFrame` and implements the `WindowListener` interface. This class, `CloseableFrame`, is shown in Figure B.10. The window close event handler is simple to write — it just calls `System.exit`. The other methods remain without a special implementation. The constructor registers that it is willing to accept the window closing event. Now we can use `CloseableFrame` instead of `JFrame` throughout.

`CloseableFrame` extends `JFrame` and implements `WindowListener`.

Notice that the code for `CloseableFrame` is cumbersome; we will revisit it shortly, and see a use for anonymous inner classes.

The `pack` method simply makes the `JFrame` as tight as possible, given its constituent components. The `show` method displays the `JFrame`.

Figure B.11 provides a `main` that can be used to start the application in Figure B.1. We place this in a separate class, which we call `BasicGUI`. `BasicGUI` extends the class `CloseableFrame`. `main` simply creates a `JFrame` into which we place a `GUI` object. We then add an unnamed `GUI` object into the `JFrame`'s content pane and `pack` the `JFrame`. The `pack` method simply makes the `JFrame` as tight as possible, given its constituent components. The `show` method displays the `JFrame`.

```
        // Handle the draw button push
    public void actionPerformed( ActionEvent evt )
    {
        try
        {
            theCanvas.setParams(
                (String) theShape.getSelectedItem( ),
                (String) theColor.getSelectedValue( ),
                Integer.parseInt( theXCoor.getText( ) ),
                Integer.parseInt( theYCoor.getText( ) ),
                smallPic.isSelected( ) ? 0 :
                        mediumPic.isSelected( ) ? 1 : 2,

                theFillBox.isSelected( ) );
            theMessage.setText( "" );
        }
        catch( NumberFormatException e )
          { theMessage.setText( "Incomplete input" ); }
    }
```

Figure B.9 Code to handle the draw button push for Figure B.1

```
// Frame that closes on a window-close event

public class CloseableFrame extends JFrame
                        implements WindowListener
{
    public CloseableFrame( )
      { addWindowListener( this ); }

    public void windowClosing( WindowEvent event )
      { System.exit( 0 ) }
    public void windowClosed( WindowEvent event )
      { }
    public void windowDeiconified( WindowEvent event )
      { }
    public void windowIconified( WindowEvent event )
      { }
    public void windowActivated( WindowEvent event )
      { }
    public void windowDeactivated( WindowEvent event )
      { }
    public void windowOpened( WindowEvent event )
      { }
}
```

Figure B.10 `CloseableFrame` class: same as `JFrame`, but handles the window closing event

```
 1 class BasicGUI extends CloseableFrame
 2 {
 3     public static void main( String [ ] args )
 4     {
 5         JFrame f = new BasicGUI( );
 6         f.setTitle( "GUI Demo" );
 7
 8         Container contentPane = f.getContentPane( );
 9         contentPane.add( new GUI( ) );
10         f.pack( );
11         f.show( );
12     }
13 }
```

Figure B.11 `main` routine for Figure B.1

B.3.4 Event Handling: Adapters and Anonymous Inner Classes

The `CloseableFrame` class is a mess. To listen for a `WindowEvent`, we must declare a class that implements the `WindowListener` interface, instantiate the class, and then register that object with the `CloseableFrame`. Since the `WindowListener` interface has seven methods, we must implement all seven methods, even though we are interested in only one of the seven methods.

One can imagine the messy code that will ensue when a large program handles numerous events. The problem is that every event handling strategy corresponds to a new class, and it would be bizarre to have many classes with lots of methods that simply declare `{ }`.

The *listener adapter classes* provide default implementations of all the listener methods.

As a result, the `java.awt.event` package defines a set of *listener adapter classes*. Each listener interface that has more than one method is implemented by a corresponding listener adapter class, with empty bodies. Thus instead of providing the empty bodies ourselves, we can simply extend the adapter class, and override the methods we are interested in. In our case, we need to extend

`WindowAdapter`. This gives the (flawed) implementation for `CloseableFrame` shown in Figure B.12.

The code in Figure B.12 fails because multiple implementation inheritance is illegal in Java. This is not a serious problem, however, because we do not need the `CloseableFrame` to be the object that handles its own events. Instead, it can delegated to a function object.

```
1 // Frame that closes on a window-close event: (flawed)
2 public class CloseableFrame extends JFrame, WindowAdapter
3 {
4     public CloseableFrame( )
5       { addWindowListener( this ); }
6
7     public void windowClosing( WindowEvent event )
8       { System.exit( 0 ) }
9 }
```

Figure B.12 `CloseableFrame` class using `WindowAdapter`. This does not work because there is no multiple inheritance in Java.

```
1 // Frame that closes on a window-close event: (works!)
2 public class CloseableFrame extends JFrame, WindowAdapter
3 {
4     public CloseableFrame( )
5       { addWindowListener( new ExitOnClose( ) ); }
6
7     private class ExitOnClose extends WindowAdapter
8     {
9         public void windowClosing( WindowEvent event )
10          { System.exit( 0 ) }
11    }
12 }
```

Figure B.13 `CloseableFrame` class using `WindowAdapter` and inner class.

```
 1 // Frame that closes on a window-close event: (works!)
 2 public class CloseableFrame extends JFrame, WindowAdapter
 3 {
 4     public CloseableFrame( )
 5     {
 6         addWindowListener( new WindowAdapter( )
 7             {
 8               public void windowClosing( WindowEvent event )
 9                 { System.exit( 0 ) }
10             }
11         );
12     }
13 }
```

Figure B.14 `CloseableFrame` class using `WindowAdapter` and anonymous inner class.

Figure B.13 illustrates this approach. The `ExitOnClose` class implements the `WindowListener` interface by extending `WindowAdapter`. An instance of that class is created and registered as the frame's window listener. `ExitOnClose` is declared as an inner class instead of a nested class. This would give it access to any of the `CloseableFrame`'s instance members, should it need it. The event handling model is a classic example of the use of function objects, and is the reason that inner classes were deemed an essential addition to the language (recall that inner classes and the new event model appeared simultaneously in Java 1.1).

Figure B.14 shows the logical continuation, using anonymous inner classes. Here we are adding a `WindowListener` and explaining, on pretty much the next line of code, what the `WindowListener` does. This is a classic use of the anonymous inner classes. The pollution of braces, parentheses and semicolons is horrific, but experienced readers of Java code skip over those syntactic details and easily see what the event handling code does. The main benefit here is that if there

are lots of small event handling methods, they need not be scattered in top-level classes, but instead can be placed near the objects that these events are coming from.

B.3.5 Summary: Putting the Pieces Together

Here is a summary of how to create a GUI application. Place the GUI functionality in a class that extends `JPanel`. For that class, do the following:

- Decide on the basic input elements and text output elements. If the same elements are used twice, make an extra class to store the common functionality and apply these principles on that class.
- If graphics are used, make an extra class that extends `JPanel`. That class must provide a `paintComponent` method and a public method that can be used by the container to communicate to it. It may also need to provide a constructor.
- Pick a layout and issue a `setLayout` command.
- Add components to the GUI using `add`.
- Handle events. The simplest way to do this is to use a `Button` and trap the button push with `actionPerformed`.

Once a GUI class is written, an application defines a class that extends `CloseableFrame` with a `main` routine. The `main` routine simply creates an instance of this extended frame class, places the GUI panel inside the frame's content pane, and issues a `pack` command and a `show` command for the frame.

B.3.6 Is This Everything I Need To Know About Swing?

What we have described so far will work well for toy user interfaces, and is an improvement over console-based applications. But there are significant complications that a professional applications programmer would have to deal with.

It is rare that the layout manager will make you happy, Often you need to tinker by adding additional subpanels. To help out, Swing defines elements such as spacers, struts, and so on that allow you to position elements more precisely, along with elaborate layout managers. Using these elements is quite challenging.

Other swing components include sliders, progress bars, scrolling (which can be added to any `JComponent`), password textfields, file choosers, option panes and dialog boxes, tree structures (such as what you see in FileManager on Windows systems), tables, and on and on. Image acquisition and display is also supported by Swing. Additionally, one often needs to know about fonts, colors, and the screen environment that one is working in.

Additionally, there is the important issue of what happens if an event occurs while you are in an event handler. It turns out that events are queued. However, if you get trapped in an event handler for a long time, you application can appear unresponsive; we've all seen this in application code. For instance, if the button handling code has an infinite loop, you will not be able to close a window. To solve this problem, typically programmers use a technique known as multithreading, which opens up a whole new can of worms.

Summary

This appendix examined the basics of the Swing package, which allows the programming of GUIs. This makes the program look much more professional than simple terminal I/O.

GUI applications differ from terminal I/O applications in that they are event-driven. To design a GUI, we write a class. We must decide on the basic input elements and output elements, pick a layout and issue a `setLayout` command, add components to the GUI using `add`, and handle events. All this is part of the class. Starting with Java 1.1, event handling is done with event listeners.

Once this class is written, an application defines a class that extends `JFrame` with a `main` routine and an event handler. The event handler processes the window closing event. The simplest way to do this is to use the `CloseableFrame` class in Figure B.14. The `main` routine simply creates an instance of this extended frame class, places an instance of the class (whose constructor likely creates a GUI panel) inside the frame's content pane, and issues a `pack` command and a `show` command for the frame.

Only the basics of Swing have been discussed here. Swing is the topic of entire books.

Objects of the Game

Abstract Window Toolkit (AWT) A GUI toolkit that is supplied with all Java systems. Provides the basic classes to allow user interfaces. (930)

`ActionEvent` An event generated when a user presses a `JButton`, hits *Return* in a `JTextField`, or selects from a `JList` or `JMenuItem`. Should be handled by the `actionPerformed` method in a class that implements the `ActionListener` interface. (956)

`ActionListener` interface An interface used to handle action events. Contains the abstract method `actionPerformed`. (956)

`actionPerformed` A method used to handle action events. (956)

`AWTEvent` An object that stores information about an event. (956)

`BorderLayout` The default for objects in the `Window` hierarchy. Used to lay out a container by placing components in one of five locations (`"North"`, `"South"`, `"East"`, `"West"`, `"Center"`). (949)

`ButtonGroup` An object used to group a collection of button objects and guarantee that only one may be *on* at any time. (944)

canvas A blank rectangular area of the screen onto which an application can draw and receive input from the user in the form of keyboard and mouse events. In Swing, this is implemented by extending `JPanel`. (953)

`Component` An abstract class that is the superclass of many AWT objects. Represents something that has a position and a size and that can be painted on the screen as well as can receive input events. (934)

`Container` The abstract superclass representing all components that can hold other components. Typically has an associated layout manager. (935)

event Produced by the operating system for various occurrences, such as input operations, and passed to Java. (956)

`FlowLayout` A layout that is the default for `JPanel`. Used to lay out a container by adding components in a row from left to right. When there is no room left in a row, a new row is formed. (948)

graphical user interface (GUI) The modern alternative to terminal I/O that allows a program to communicate with its user via buttons, checkboxes, textfields, choice lists, menus, and the mouse. (929)

`Graphics` An abstract class that defines several methods that can be used to draw shapes. (953)

`JButton` A component used to create a labeled button. When the button is pushed, an action event is generated. (940)

`JCheckBox` A component that has an *on* state and an *off* state. (944)

`JComboBox` A component used to select a single string via a pop-up list of choices. (941)

`JComponent` An abstract class that is the superclass of lightweight Swing objects. (936)

`JDialog` A top-level window used to create dialogs. (936)

`JFrame` A top-level window that has a border and can also have an associated `JMenuBar`. (936)

`JLabel` A component that is used to label other components such as a `JComboBox`, `JList`, `JTextField`, or `JPanel`. (938)

`JList` A component that allows the selection from a scrolling list of strings. Can allow one or multiple selected items, but uses more screen real estate than `JComboBox`. (943)

`JPanel` A container used to store a collection of objects but does not create borders. Also used for canvasses. (937)

`JTextArea` A component that presents the user with several lines of text. (946)

`JTextField` A component that presents the user with a single line of text. (946)

layout manager A helper object that automatically arranges components of a container. (947)

listener adapter class Provides default implementations for a listener interface that has more than one method. (960)

`null` layout A layout used to perform precise positioning. Allows the `setBounds` method to work. (951)

`pack` A method used to pack a `JFrame` into its smallest size given its constituent components. (958)

`paintComponent` A method used to draw onto a component. Typically overridden by classes that extend `JPanel`. (953)

`repaint` A method used to clear and repaint a component. (954)

`setLayout` A method that associates a layout with a container. (947)

`show` A method that makes a component visible. (958)

`Window` A top-level window that has no border. (936)

`WindowAdapter` A class that provides default implementations of the `WindowListener` interface. (936)

`WindowListener` interface An interface used to specify the handling of window events, such as window closing. (958)

Common Errors

1. Forgetting to set a layout manager is a common mistake. If you forget it, you'll get a default. However, it may not be the one you want.
2. The layout manager must appear prior to the calls to `add`.
3. Applying `add` or setting a layout manager to the wrong container is a common mistake. For instance, in a container that contains panels, applying the `add` method without specifying the panel means that the `add` is applied to the main container.
4. A missing `String` argument to `add` for `BorderLayout` uses `"Center"` as the default. A common mistake is to specify it in the wrong case, as in `"north"`. The five valid arguments are `"North"`, `"South"`, `"East"`, `"West"`, and `"Center"`. In Java 1.1, if the `String` is the second parameter, a runtime exception will catch the error. If you use the old style, in which the `String` comes first, the error might not be detected.
5. Special code is needed to process the window closing event.

On the Internet

All code found in this Appendix is available:

BorderTest.java	Simple illustration of the `BorderLayout`, shown in Figure B.6.
BasicGUI.java	The main example, for the GUI application used in this chapter, with `CloseableFrame` from Figure B.14.

Exercises

In Short

B.1. What is a GUI?

B.2. List the various `JComponent` classes that can be used for GUI input.

B.3. Describe the difference between heavyweight components and lightweight components, and give examples of each.

B.4. What are the differences between the `JList` and `JComboBox` components?

B.5. What is a `ButtonGroup` used for?

B.6. Explain the steps taken to design a GUI.

B.7. Explain how the `FlowLayout`, `BorderLayout`, and `null` layouts arrange components.

B.8. Describe the steps taken to include a graphical component inside a `JPanel`.

B.9. What is the default behavior when an event occurs? How is the default changed?

B.10. What events generate an `ActionEvent`?

B.11. How is the window closing event handled?

In Practice

B.12. `paintComponent` can be written for any component. Show what happens when a circle is painted in the GUI class instead of its own canvas.

B.13. Handle the pressing of the Enter key in the *y*-coordinate text field in class `GUI`. You will need to modify `actionPerformed` and register a second event handler.

B.14. Add a default of (0, 0) for the coordinates of a shape in class `GUI`.

Programming Projects

B.15. Write a program that can be used to input two dates and output the number of days between them. Use the `Date` class from Exercise 3.16.

B.16. Write a program that allows you to draw lines inside a canvas using the mouse. A click starts the line draw; a second click ends the line. Multiple lines can be drawn on the canvas. To do this, extend the `JPanel` class and handle mouse events by implementing `MouseListener`. You will also need to override `update` to avoid clearing the canvas between line draws. Add a button to clear the canvas.

B.17. Write an application that contains two `GUI` objects. When actions occur in one of the `GUI` objects, the other `GUI` object saves its old state. You will

need to add a `copyState` method to the `GUI` class that will copy the states of all of the `GUI` fields and redraw the canvas.

B.18. Write a program that contains a single canvas and a set of ten GUI input components that each specify a shape, color, coordinates, and size, and a checkbox that indicates the component is active. Then draw the union of the input components onto a canvas. Represent the GUI input component by using a class with accessor functions. The main program should have an array of these input components plus the canvas.

Reference

In addition to the standard set of references in Chapter 1, a complete Swing tutorial is provided in the 950 page book [1].

1. K. Walrath and M. Campione, *The JFC Swing Tutorial*, Addison-Wesley, Reading, Mass. (1999).