CHAPTER 18

# Swing II

# 18 Swing II

*Window listeners?*
*I thought windows were for looking not for listening.*

Student answer on an exam

### Introduction

Although this is the third chapter on Swing, we have entitled it "Swing II" because Chapter 16, entitled "Swing I," and this chapter are really one unit. This chapter is a continuation of Chapter 16, presenting more details about designing regular Swing GUIs. Chapter 17 on applets is a side issue that may be read after this chapter if you prefer.

### Prerequisites

This chapter uses material from Chapter 16 (and its prerequisites).

Section 18.2 on icons and scroll bars is not used in subsequent sections and so may be skipped or postponed.

## 18.1 Window Listeners

*A man may see how this world goes with no eyes.*
*Look with thine ears. . . .*

William Shakespeare, *King Lear*

In Chapter 16 we used the method `setDefaultCloseOperation` to program the close-window button in a `JFrame`. This allows for only a limited number of possibilities for what happens when the close-window button is clicked. When the user clicks the close-window button (or either of the two accompanying buttons), the `JFrame` fires an event known as a **window event**. A `JFrame` can use the method `setWindowListener` to register a **window listener** to respond to such window events. A window listener can be programmed to respond to a window event in any way you wish. Window events are objects of the class `WindowEvent`. A window listener is any class that satisfies the `WindowListener` interface.

window event
window listener
WindowEvent
WindowListener

The method headings in the `WindowListener` interface are given in Display 18.1. If a class implements the `WindowListener` interface, it must have definitions for all seven of these method headings. If you do not need all of these methods, then you can define the ones you do not need to have empty bodies, like this:

```
public void windowDeiconified(WindowEvent e)
```

{}.

**Display 18.1 Methods in the `WindowListener` Interface**

The `WindowListener` interface and the `WindowEvent` class are in the package `java.awt.event`.

```
public void windowOpened(WindowEvent e)
```

Invoked when a window has been opened.

```
public void windowClosing(WindowEvent e)
```

Invoked when a window is in the process of being closed. Clicking the close-window button causes an invocation of this method.

```
public void windowClosed(WindowEvent e)
```

Invoked when a window has been closed.

```
public void windowIconified(WindowEvent e)
```

Invoked when a window is iconified. When you click the minimize button in a `JFrame`, it is iconified.

```
public void windowDeiconified(WindowEvent e)
```

Invoked when a window is deiconified. When you activate a minimized window, it is deiconified.

```
public void windowActivated(WindowEvent e)
```

Invoked when a window is activated. When you click in a window, it becomes the activated window. Other actions can also activate a window.

```
public void windowDeactivated(WindowEvent e)
```

Invoked when a window is deactivated. When a window is activated, all other windows are deactivated. Other actions can also deactivate a window.

### The `WindowListener` Interface

When the user clicks any of the three standard `JFrame` buttons (for closing the window, minimizing the window, and resizing the window), that generates a window event. Window events are sent to window listeners. In order to be a window listener, a class must implement the `WindowListener` interface. The method headings for the `WindowListener` interface are given in Display 18.1.

**Example**

### A Window Listener Inner Class

Display 18.2 gives an example of a JFrame class with a window listener class that is an inner class. The window listener inner class is named CheckOnExit. A window listener class need not be an inner class, but it is frequently convenient to make a window listener class (or other kind of listener class) an inner class.

The main JFrame in Display 18.2 simply displays a message. What is interesting is how the window listener programs the close-window button. You can apply the window listener used in this JFrame to any JFrame. When the close-window button is clicked, a second, smaller window appears and asks: "Are you sure you want to exit?". If the user clicks the "Yes" button, the entire program ends and so both windows go away. If the user clicks the "No" button, only the smaller window disappears; the program and the main window continue. Let's look at the programming details.

When the close-window button in the main window is clicked, that fires a window event. The only registered window listener is the anonymous object that is the argument to addWindowListener. Below we repeat the relevant line of code, which is in the constructor for WindowListenerDemo:

```
addWindowListener(new CheckOnExit());
```

This anonymous window listener object receives the window event fired when the close-window button is clicked and then invokes the method windowClosing. The method windowClosing creates and displays a window object of the class ConfirmWindow, which contains the message "Are you sure you want to exit?" as well as the two buttons labeled "Yes" and "No".

If the user clicks the "Yes" button, the action event fired by that button goes to the actionPerformed method, which ends the program with a call to System.exit. If the user clicks the "No" button, then the actionPerformed method invokes the method dispose. The method dispose, discussed in the next subsection, makes its calling object go away but does not end the program. The calling object for dispose is the smaller window (which is an object of the class ConfirmWindow), so this smaller window goes away but the main window remains.

setDefaultCloseOperation

Notice that even though we have registered a window listener, which says what should happen when the close-window button is clicked, we still need to invoke the method setDefaultCloseOperation. When the close-window button is clicked, the policy set by setDefaultCloseOperation is always carried out in addition to any action by window listeners. If we do not include any invocation of setDefaultCloseOperation, then the default action is to make the window go away (but not to end the program). We do not want our main window to go away, so set the policy as follows:

```
setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

This means that clicking the close-window button causes no action other than the actions of any window listeners. If you are using a window listener to set the action of the close-window button, you invariably want an invocation of setDefaultCloseOperation with the argument JFrame.DO_NOTHING_ON_CLOSE.

**Display 18.2 A Window Listener *(Part 1 of 3)***

```
1  import javax.swing.JFrame;
2  import javax.swing.JPanel;
3  import java.awt.Container;
4  import java.awt.BorderLayout;
5  import java.awt.FlowLayout;
6  import java.awt.Color;
7  import javax.swing.JLabel;
8  import javax.swing.JButton;
9  import java.awt.event.ActionListener;
10 import java.awt.event.ActionEvent;
11 import java.awt.event.WindowListener;
12 import java.awt.event.WindowEvent;

13 public class WindowListenerDemo extends JFrame
14 {
15     public static final int WIDTH = 300; //for main window
16     public static final int HEIGHT = 200; //for main window
17     public static final int SMALL_WIDTH = 200; //for confirm window
18     public static final int SMALL_HEIGHT = 100;//for confirm window

19     private class CheckOnExit implements WindowListener
20     {
21         public void windowOpened(WindowEvent e)
22         {}

23         public void windowClosing(WindowEvent e)
24         {
25             ConfirmWindow checkers = new ConfirmWindow();
26             checkers.setVisible(true);
27         }

28         public void windowClosed(WindowEvent e)
29         {}

30         public void windowIconified(WindowEvent e)
31         {}

32          public void windowDeiconified(WindowEvent e)
33         {}

34         public void windowActivated(WindowEvent e)
35         {}
```

This `WindowListener` class is an inner class.

*A window listener must define all the method headings in the `WindowListener` interface, even if some are trivial implementations.*

**Display 18.2 A Window Listener *(Part 2 of 3)***

```
36          public void windowDeactivated(WindowEvent e)
37          {}
38      } //End of inner class CheckOnExit

39      private class ConfirmWindow extends JFrame implements ActionListener
40      {
41          public ConfirmWindow()
42          {
43              setSize(SMALL_WIDTH, SMALL_HEIGHT);
44              Container confirmContent = getContentPane();
45              confirmContent.setBackground(Color.YELLOW);
46              confirmContent.setLayout(new BorderLayout());

47              JLabel confirmLabel = new JLabel(
48                              "Are you sure you want to exit?");
49              confirmContent.add(confirmLabel, BorderLayout.CENTER);

50              JPanel buttonPanel = new JPanel();
51              buttonPanel.setBackground(Color.ORANGE);
52              buttonPanel.setLayout(new FlowLayout());

53              JButton exitButton = new JButton("Yes");
54              exitButton.addActionListener(this);
55              buttonPanel.add(exitButton);

56              JButton cancelButton = new JButton("No");
57              cancelButton.addActionListener(this);
58              buttonPanel.add(cancelButton);

59              confirmContent.add(buttonPanel, BorderLayout.SOUTH);
60          }

61          public void actionPerformed(ActionEvent e)
62          {
63              String actionCommand = e.getActionCommand();

64              if (actionCommand.equals("Yes"))
65                  System.exit(0);
66              else if (actionCommand.equals("No"))
67                  dispose();//Destroys only the ConfirmWindow.
68              else
69                  System.out.println("Unexpected Error in Confirm Window.");
70          }
71      } //End of inner class ConfirmWindow
```

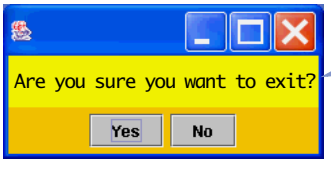
Another inner class.

**Display 18.2** **A Window Listener *(Part 3 of 3)***

```
72
73      public static void main(String[] args)
74      {
75          WindowListenerDemo demoWindow = new WindowListenerDemo();
76          demoWindow.setVisible(true);
77      }
78
79      public WindowListenerDemo()
80      {
81          setSize(WIDTH, HEIGHT);
82          setTitle("Window Listener Demonstration");
83
84          setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
85          addWindowListener(new CheckOnExit());
86
87          Container contentPane = getContentPane();
88          contentPane.setBackground(Color.LIGHT_GRAY);
89          JLabel aLabel = new JLabel("I like to be sure you are sincere.");
90          contentPane.add(aLabel);
91      }
92  }
```

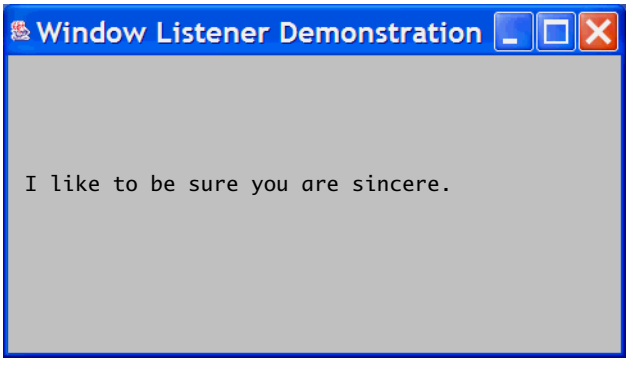
Even if you have a window listener, you normally must still invoke `setDefaultCloseOperation`.

### Resulting GUI




This window is an object of the class `ConfirmWindow`.

When you click this close-window button, the second window appears.



## THE dispose METHOD

dispose

The method `dispose` is a method in the class `JFrame` that releases any resources used by the `JFrame` or any of its components. So, a call to `dispose` eliminates the `JFrame` and its components, but if the program has items that are not components of the `JFrame`, then the program does not end. For example, in Display 18.2 the smaller window of the class `ConfirmWindow` invokes `dispose` (if the user clicks the `"No"` button). That causes that smaller window to go away, but the larger window remains.

### The dispose Method

The class `JFrame` has a method named `dispose` that will eliminate the invoking `JFrame` without ending the program. When `dispose` is invoked, the resources consumed by the `JFrame` and its components are returned for reuse, so the `JFrame` is gone, but the program does not end (unless `dispose` eliminates all elements in the program, as in a one-window program). The method `dispose` is often used in a program with multiple windows to eliminate one window without ending the program.

**Syntax:**

```
JFrame_Object.dispose();
```

The *JFrame_Object* is often an implicit `this`. A complete example of using `dispose` can be seen in Display 18.2.

**Pitfall**

### Forgetting to Invoke setDefaultCloseOperation

If you register a window listener to respond to window events from a `JFrame`, you should also include an invocation of the method `setDefaultCloseOperation`, typically in the `JFrame` constructor. This is because the behavior set by `setDefaultCloseOperation` takes place even if there is a window listener. If you do not want any actions other than those provided by the window listener(s), you should include the following in the `JFrame` constructor:

```
setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

If you do not include any invocation of `setDefaultCloseOperation`, the default action is the same as if you had included the following invocation

```
setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
```

which hides the `JFrame` when the close-window button is clicked. (The actions of any registered window listener are also performed.)

## THE WindowAdapter CLASS

In Display 18.2 we gave empty bodies to most of the method headings in the `WindowListener` interface. The abstract class `WindowAdapter` is a way to avoid all those empty method bodies. The class `WindowAdapter` does little more than provide trivial implementations of the method headings in the `WindowListener` interface. So, if you make a window listener a derived class of the class `WindowAdapter`, then you only need to define the method headings in the `WindowListener` interface that you need. The other method headings inherit trivial implementations from `WindowAdapter`. (`WindowAdapter` is unusual in that it is an abstract class with no abstract methods.)

For example, in Display 18.3 we have rewritten the inner class `CheckOnExit` from Display 18.2, but this time we made it a derived class of the `WindowAdapter` class. This definition of `CheckOnExit` is much shorter and cleaner than the one in Display 18.2, but the two implementations of `CheckOnExit` are equivalent. Thus, you can replace the definition of `CheckOnExit` in Display 18.2 with the shorter one in Display 18.3. The file `WindowListenerDemo2` on the accompanying CD contains a version of Display 18.2 with this shorter definition of `CheckOnExit`.

extra code on CD

The class `WindowAdapter` is in the `java.awt.event` package and so requires an `import` statement such as the following:

```
import java.awt.event.WindowAdapter;
```

You cannot always define your window listeners as derived classes of `WindowAdapter`. For example, suppose you want a `JFrame` class to be its own window listener. To accomplish that, the class must be a derived class of `JFrame` and so cannot be a derived class of any other class such as `WindowAdapter`. In such cases, you make the class a derived class of `JFrame` and have it implement the `WindowListener` interface. See Self-Test Exercise 4 for an example.

**Display 18.3 Using `WindowAdapter`**

*This requires the following import statements:*

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
```

```
1  private class CheckOnExit extends WindowAdapter
2  {
3      public void windowClosing(WindowEvent e)
4      {
5          ConfirmWindow checkers = new ConfirmWindow();
6          checkers.setVisible(true);
7      }
8  } //End of inner class CheckOnExit
```

*If the definition of the inner class `CheckOnExit` in Display 18.2 were replaced with this definition of `CheckOnExit`, there would be no difference in how the outer class or any class behaves.*

### Self-Test Exercises

1. When you define a class and make it implement the `WindowListener` interface, what methods must you define? What do you do if there is no particular action that you want one of these methods to take?
2. The GUI in Display 18.2 has a main window. When the user clicks the close-window button in the main window, a smaller window appears that says `"Are you sure you want to exit?"`. What happens if the user clicks the close-window button in this smaller window? Explain your answer.
3. If you want a Swing program to end completely, you can invoke the method `System.exit`. What if you want a `JFrame` window to go away, but you do not want the program to end? What method can you have the `JFrame` invoke?
4. Rewrite the class in Display 18.2 so that the class is its own window listener. Hint: The constructor will contain

```
addWindowListener(this);
```

## 18.2 Icons and Scroll Bars

I ♥ ICONS.

Bumper sticker

### ICONS

icon

`JLabel`s, `JButton`s, and `JMenuItem`s can have icons. An **icon** is simply a small picture, although it is not required to be small. The label, button, or menu item may have just a string displayed on it, just an icon, or both (or it can have nothing at all on it). An icon is an instance of the `ImageIcon` class and is based on a digital picture file. The picture file can be in almost any standard format, such as `.gif`, `.jpg`, or `.tiff`.

ImageIcon

The class `ImageIcon` is used to convert a picture file to a Swing icon. For example, if you have a picture in a file named `duke_waving.gif`, the following will produce an icon named `dukeWavingIcon` for the picture `duke_waving.gif`:

```
ImageIcon dukeIcon = new ImageIcon("duke_waving.gif");
```

The file `duke_waving.gif` should be in the same directory as the class in which this code appears. Alternatively, you can use a complete or relative path name to specify the picture file. Note that the picture file is given as a value of type `String` that names the picture file. The file `duke_waving.gif` and other picture files we will use in this chapter are all provided on the CD that accompanies this text.

You can add an icon to a label with the method `setIcon`, as follows: setIcon

```
JLabel dukeLabel = new JLabel("Mood check");
dukeLabel.setIcon(dukeIcon);
```

Alternatively, you give the icon as an argument to the `JLabel` constructor, as follows:

```
JLabel dukeLabel = new JLabel(dukeIcon);
```

You can leave the label as created and it will have an icon but no text, or you can add text with the method `setText`, as follows: setText

```
dukeLabel.setText("Mood check");
```

Icons and text may be added to `JButton`s and `JMenuItem`s in the same way as they are added to a `JLabel`. For example, the following is taken from Display 18.4, which is a demonstration of the use of icons:

```
JButton happyButton = new JButton("Happy");
ImageIcon happyIcon = new ImageIcon("smiley.gif");
happyButton.setIcon(happyIcon);
```

**Display 18.4 Using Icons *(Part 1 of 3)***

```
 1  import javax.swing.JFrame;
 2  import javax.swing.JPanel;
 3  import javax.swing.JTextField;
 4  import javax.swing.ImageIcon;
 5  import java.awt.Container;
 6  import java.awt.BorderLayout;
 7  import java.awt.FlowLayout;
 8  import java.awt.Color;
 9  import javax.swing.JLabel;
10  import javax.swing.JButton;
11  import java.awt.event.ActionListener;
12  import java.awt.event.ActionEvent;


13  public class IconDemo extends JFrame implements ActionListener
14  {
15      public static final int WIDTH = 500;
16      public static final int HEIGHT = 200;
17      public static final int TEXT_FIELD_SIZE = 30;

18      private JTextField message;
```

**Display 18.4 Using Icons *(Part 2 of 3)***

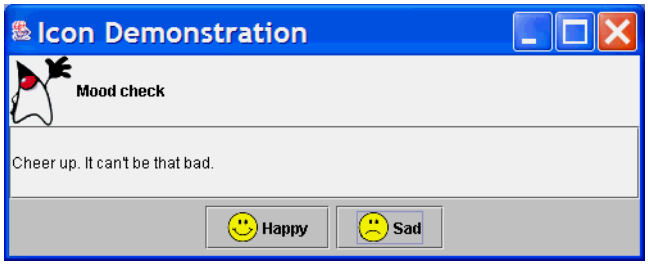
```
19      public static void main(String[] args)
20      {
21          IconDemo iconGui = new IconDemo();
22          iconGui.setVisible(true);
23      }


24      public IconDemo()
25      {
26          super("Icon Demonstration");
27          setSize(WIDTH, HEIGHT);
28          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

29          Container contentPane = getContentPane();
30          contentPane.setBackground(Color.WHITE);
31          contentPane.setLayout(new BorderLayout());

32          JLabel dukeLabel = new JLabel("Mood check");
33          ImageIcon dukeIcon = new ImageIcon("duke_waving.gif");
34          dukeLabel.setIcon(dukeIcon);
35          contentPane.add(dukeLabel, BorderLayout.NORTH);

36          JPanel buttonPanel = new JPanel();
37          buttonPanel.setLayout(new FlowLayout());
38          JButton happyButton = new JButton("Happy");
39          ImageIcon happyIcon = new ImageIcon("smiley.gif");
40          happyButton.setIcon(happyIcon);
41          happyButton.addActionListener(this);
42          buttonPanel.add(happyButton);
43          JButton sadButton = new JButton("Sad");
44          ImageIcon sadIcon = new ImageIcon("sad.gif");
45          sadButton.setIcon(sadIcon);
46          sadButton.addActionListener(this);
47          buttonPanel.add(sadButton);
48          contentPane.add(buttonPanel, BorderLayout.SOUTH);

49          message = new JTextField(TEXT_FIELD_SIZE);
50          contentPane.add(message, BorderLayout.CENTER);
51      }


52      public void actionPerformed(ActionEvent e)
53      {
54         String actionCommand = e.getActionCommand();
```

**Display 18.4 Using Icons *(Part 3 of 3)***

```
55          if (actionCommand.equals("Happy"))
56              message.setText(
57                      "Smile and the world smiles with you!");
58          else if (actionCommand.equals("Sad"))
59              message.setText(
60                     "Cheer up. It can't be that bad.");
61          else
62               message.setText("Unexpected Error.");
63      }
64  }
```

**RESULTING GUI[1]**

*View after clicking the "Sad" button.*




button with only an icon

You can produce a button or menu item with (just) an icon on it by giving the `ImageIcon` object as an argument to the `JButton` or `JMenuItem` constructor. For example:

```
ImageIcon happyIcon = new ImageIcon("smiley.gif");
JButton smileButton = new JButton(happyIcon);
JMenuItem happyChoice = new JMenuItem(happyIcon);
```

If you create a button or menu item in this way and do not add text with the method `setText`, you should use `setActionCommand` to explicitly give the button or menu item an action command, since there is no string on the button or menu item.

All of the classes `JButton`, `JMenuItem`, and `JLabel` have constructors that let you specify text and an icon to appear on the button, menu item, or label. The constructor can specify no text or icon, text only, an icon only, or both text and an icon. When you specify both text and an icon, the text is the first argument and the icon is the second argument; also, the constructor for a `JLabel` requires a third argument, as described in

[1] Java, Duke, and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

### Icons and the Class ImageIcon

An **icon** is simply a small picture, although it is not really required to be small. The class ImageIcon is used to convert a picture file to a Swing icon.

**Syntax:**

```
ImageIcon Name_Of_ImageIcon =
        new ImageIcon(Picture_File_Name);
```

The *Picture_File_Name* is a string giving either a relative or absolute path name to the picture file. (So if the picture file is in the same directory as your program, you need give only the name of the picture file.)

**Example:**

```
ImageIcon happyIcon =
        new ImageIcon("smiley.gif");
```

### setIcon and setText

The method setIcon can be used to add an icon to a JButton, JMenuItem, or JLabel. The argument to setIcon must be an ImageIcon object.

**Syntax:**

```
Component.setIcon(ImageIcon_Object);
```

The *Component* can be a JButton, JMenuItem, or JLabel.

**Example:**

```
JLabel helloLabel = new JLabel("Hello");
ImageIcon dukeIcon = new ImageIcon("duke_waving.gif");
helloLabel.setIcon(dukeIcon);
```

The method setText can be used to add text to a JButton, JMenuItem, or JLabel.

**Syntax:**

```
Component.setText(Text_String);
```

The *Component* can be a JButton, JMenuItem, or JLabel.

**Example:**

```
ImageIcon dukeIcon = new ImageIcon("duke_waving.gif");
JLabel helloLabel = new JLabel(dukeIcon);
helloLabel.setText("Hello");
```

The two examples are equivalent.

Display 18.5. If you omit either text or an icon (or both) from the constructor, you can add them later with the methods `setText` and `setIcon`. Some of these methods for the classes `JButton`, `JMenuItem`, and `Jlabel` are given in Display 18.5.

**Display 18.5 Some Methods in the Classes `JButton`, `JMenuItem`, and `JLabel` *(Part 1 of 2)***

```
public JButton()
public JMenuItem()
public JLabel()
```

Creates a button, menu item, or label with no text or icon on it. (Typically, you will later use `setText` and/or `setIcon` with the button, menu item, or label.)

```
public JButton(String text)
public JMenuItem(String text)
public JLabel(String text)
```

Creates a button, menu item, or label with the `text` on it.

```
public JButton(ImageIcon picture)
public JMenuItem(ImageIcon picture)
public JLabel(ImageIcon picture)
```

Creates a button, menu item, or label with the icon `picture` on it and no text.

```
public JButton(String text, ImageIcon picture)
public JMenuItem(String text, ImageIcon picture)
public JLabel(
       String text, ImageIcon picture, int horizontalAlignment)
```

Creates a button, menu item, or label with both the `text` and the icon `picture` on it. `horizontal-Alignment` is one of the constants `SwingConstants.LEFT`, `SwingConstants.CENTER`, `SwingCon-stants.RIGHT`, `SwingConstants.LEADING`, or `SwingConstants.TRAILING`.

The interface `SwingConstants` is in the `javax.swing` package.

```
public void setText(String text)
```

Makes `text` the only text on the button, menu item, or label.

```
public void setIcon(ImageIcon picture)
```

Makes `picture` the only icon on the button, menu item, or label.

```
public void setMargin(Insets margin)
```

`JButton` and `JMenuItem` have the method `setMargin`, but `JLabel` does not.

The method `setMargin` sets the size of the margin around the text and icon in the button or menu item. The following special case will work for most simple situations. The `int` values give the number of pixels from the edge of the button or menu item to the text and/or icon.

```
public void setMargin(new Insets(
                 int top, int left, int bottom, int right))
```

The class `Insets` is in the `java.awt` package. (We will not be discussing any other uses for the class `Insets`.)

**Display 18.5 Some Methods in the Classes `JButton`, `JMenuItem`, and `JLabel` *(Part 2 of 2)***

```
public void setVerticalTextPosition(int textPosition)
```

Sets the vertical position of the text relative to the icon. The `textPosition` should be one of the constants `SwingConstants.TOP`, `SwingConstants.CENTER` (the default position), or `SwingConstants.BOTTOM`.

The interface `SwingConstants` is in the `javax.swing` package.

```
public void setHorizontalTextPosition(int textPosition)
```

Sets the horizontal position of the text relative to the icon. The `textPosition` should be one of the constants `SwingConstants.RIGHT`, `SwingConstants.LEFT`, `SwingConstants.CENTER`, `SwingConstants.LEADING`, or `SwingConstants.TRAILING`.

The interface `SwingConstants` is in the `javax.swing` package.

### The `Insets` Class

Objects of the class `Insets` are used to specify the size of the margin in a button or menu item. The `Insets` class is in the package `java.awt`. The parameters in the following constructors are in pixels.

#### Constructor:

```
public Insets(int top, int left, int bottom, int right)
```

#### Examples:

```
aButton.setMargin(new Insets(10, 20, 10, 20));
```

## Self-Test Exercises

5. Write code to create a button that has on it both the text `"Magic Button"` and the picture in the file `wizard.gif`.
6. Write code to add the picture in the file `wizard.gif` to the `JPanel` named `picturePanel`. Assume that `picturePanel` has a `FlowLayout` manager.
7. Suppose you want to create a button that has the picture in the file `wizard.gif` on it and no text. Suppose further that you want the button to have the action command `"Kazam"`. How would you create the button and set up the action command?

## SCROLL BARS

When you create a text area, you specify the number of lines that are visible and the number of characters per line, as in the following example:

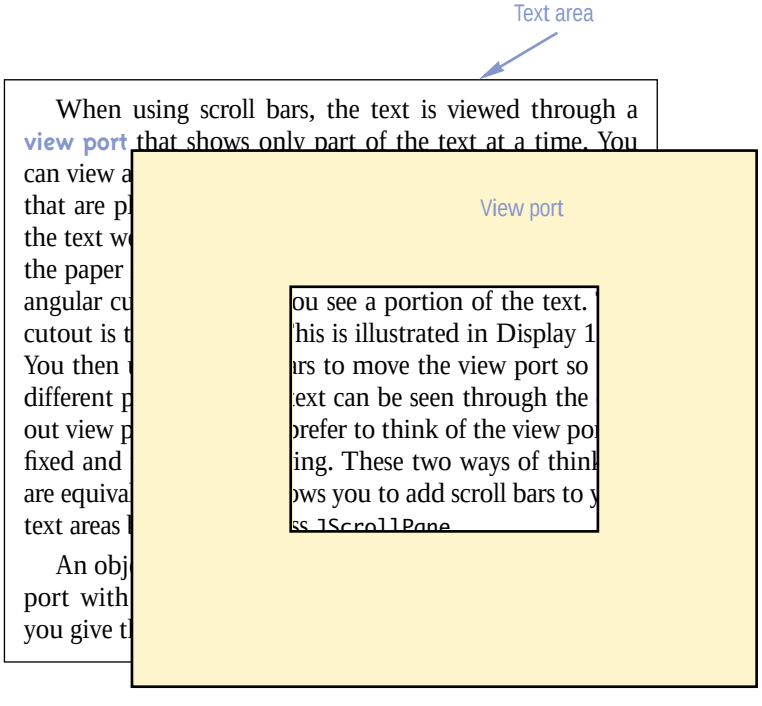
```
JTextArea memoDisplay = new JTextArea(15, 30);
```

The text area `memoDisplay` will have room for 15 lines of text, and each line will have room for at least 30 characters. The user can enter more text, but only a limited amount of text will be visible. It would be better not to have a firm limit on the number of lines or the number of characters per line that the user can see in some convenient way. The way to accomplish this is to add scroll bars to the text area, although, as you will see, the Java code looks more like adding the text area to the scroll bars rather than the other way round.

When using scroll bars, the text is viewed through a **view port** that shows only part of the text at a time. You can view a different part of the text by using the scroll bars that are placed along the side and bottom of the view port. It is as if the text were written on an unbounded sheet of paper, but the paper is covered by another piece of paper with a rectangular cutout that lets you see only a portion of the text. The cutout is the view port. This is illustrated in Display 18.6. You use the scroll bars to move the view

view port

**Display 18.6 View Port for a Text Area**



port so that different portions of the text can be seen through the cutout view port. (You may prefer to think of the view port as fixed and the text as moving. These two ways of thinking are equivalent.) Swing allows you to add scroll bars to your text areas by using the class `JScrollPane`.

JScroll-Pane

An object of the class `JScrollPane` is essentially a view port with scroll bars. When you create a `JScrollPane`, you give the text area as an argument to the `JScrollPane` constructor. For example, if `memoDisplay` is an object of the class `JTextArea` (as created in the line of code at the start of this subsection), you can place `memoDisplay` in a `JScrollPane` as follows:

```
JScrollPane scrolledText = new JScrollPane(memoDisplay);
```

The `JScrollPane` can then be added to a container, such as a `JPanel` or `JFrame`, as follows:

```
textPanel.add(scrolledText);
```

This is illustrated by the program in Display 18.8.

Note the following two lines in the constructor definition in Display 18.8:

setting scroll bar policies

```
scrolledText.setHorizontalScrollBarPolicy(
                JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
scrolledText.setVerticalScrollBarPolicy(
                JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
```

Despite the imposing length of these two method invocations, they perform a very simple task. The first merely specifies that the horizontal scroll bar will always be present. The second specifies that the vertical scroll bar will always be present.

### Scroll Bars

The class `JScrollPane` is used to add scroll bars to a `JTextArea` (and certain other components). The `JTextArea` object is given as an argument to the constructor that creates the `JScrollPane`. The `JScrollPane` class is in the `javax.swing` package.

### Syntax:

```
JScrollPane Identifier = new JScrollPane(Text_Area_Object);
```

### Examples:

```
JTextArea memoDisplay = new JTextArea(LINES, CHAR_PER_LINE);
JScrollPane scrolledText = new JScrollPane(memoDisplay);
textPanel.add(scrolledText);
```

If you omit the invocation of the two methods `setHorizontalScrollBarPolicy` and `setVerticalScrollBarPolicy`, the scroll bars will be visible only when you need them. In other words, if you omit these two method invocations and all the text fits in the view port, then no scroll bars will be visible. When you add enough text to need scroll bars, the needed scroll bars will appear automatically.

Display 18.7 summarizes what we have said about the class `JScrollPane`. We are interested in using `JScrollPane` only with text areas. However, as we note in Display 18.7, `JScrollPane` can be used with almost any sort of component.

## Self-Test Exercises

8. When setting up a `JScrollPane`, do you have to invoke both of the methods `setHorizontalScrollBarPolicy` and `setVerticalScrollBarPolicy`?

**Display 18.7 Some Methods in the Class `JScrollPane`**

The `JScrollPane` class is in the `javax.swing` package.

```
public JScrollPane(Component objectToBeScrolled)
```

Creates a new `JScrollPane` for the `objectToBeScrolled`. Note that the `objectToBeScrolled` need not be a `JTextArea`, although that is the only type of argument considered in this book.

```
public void setHorizontalScrollBarPolicy(int policy)
```

Sets the policy for showing the horizontal scroll bar. The `policy` should be one of

```
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
```

The phrase AS_NEEDED means the scroll bar is shown only when it is needed. This is explained more fully in the text. The meanings of the other policy constants are obvious from their names.

(As indicated, these constants are defined in the class `JScrollPane`. You should not need to even be aware of the fact that they have `int` values. Think of them as policies, not as `int` values.)

```
public void setVerticalScrollBarPolicy(int policy)
```

Sets the policy for showing the vertical scroll bar. The `policy` should be one of

```
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS
JScrollPane.VERTICAL_SCROLLBAR_NEVER
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED
```

The phrase AS_NEEDED means the scroll bar is shown only when it is needed. This is explained more fully in the text. The meanings of the other policy constants are obvious from their names.

(As indicated, these constants are defined in the class `JScrollPane`. You should not need to even be aware of the fact that they have `int` values. Think of them as policies, not as `int` values.)

**Display 18.8 A Text Area with Scroll Bars *(Part 1 of 3)***

```
1  import javax.swing.JFrame;
2  import javax.swing.JTextArea;
3  import javax.swing.JPanel;
4  import javax.swing.JLabel;
5  import javax.swing.JButton;
6  import javax.swing.JScrollPane;
7  import java.awt.Container;
8  import java.awt.BorderLayout;
9  import java.awt.FlowLayout;
10 import java.awt.Color;
11 import java.awt.event.ActionListener;
12 import java.awt.event.ActionEvent;


13 public class ScrollBarDemo extends JFrame
14                            implements ActionListener
15 {
16     public static final int WIDTH = 600;
17     public static final int HEIGHT = 400;
18     public static final int LINES = 15;
19     public static final int CHAR_PER_LINE = 30;


20     private JTextArea memoDisplay;
21     private String memo1;
22     private String memo2;


23     public static void main(String[] args)
24     {
25         ScrollBarDemo gui = new ScrollBarDemo();
26         gui.setVisible(true);
27     }


28     public ScrollBarDemo()
29     {
30         super("Scroll Bars Demo");
31         setSize(WIDTH, HEIGHT);
32         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
33         Container contentPane = getContentPane();
```

**Display 18.8 A Text Area with Scroll Bars *(Part 2 of 3)***

```
34          JPanel buttonPanel = new JPanel();
35          buttonPanel.setBackground(Color.LIGHT_GRAY);
36          buttonPanel.setLayout(new FlowLayout());
37          JButton memo1Button = new JButton("Save Memo 1");
38          memo1Button.addActionListener(this);
39          buttonPanel.add(memo1Button);

40          JButton memo2Button = new JButton("Save Memo 2");
41          memo2Button.addActionListener(this);
42          buttonPanel.add(memo2Button);

43          JButton clearButton = new JButton("Clear");
44          clearButton.addActionListener(this);
45          buttonPanel.add(clearButton);

46          JButton get1Button = new JButton("Get Memo 1");
47          get1Button.addActionListener(this);
48          buttonPanel.add(get1Button);

49          JButton get2Button = new JButton("Get Memo 2");
50          get2Button.addActionListener(this);
51          buttonPanel.add(get2Button);

52          contentPane.add(buttonPanel, BorderLayout.SOUTH);

53          JPanel textPanel = new JPanel();
54          textPanel.setBackground(Color.BLUE);

55          memoDisplay = new JTextArea(LINES, CHAR_PER_LINE);
56          memoDisplay.setBackground(Color.WHITE);

57          JScrollPane scrolledText = new JScrollPane(memoDisplay);
58          scrolledText.setHorizontalScrollBarPolicy(
59                      JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
60          scrolledText.setVerticalScrollBarPolicy(
61                      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

62          textPanel.add(scrolledText);

63          contentPane.add(textPanel, BorderLayout.CENTER);
64      }
```

**Display 18.8 A Text Area with Scroll Bars *(Part 3 of 3)***

```
65      public void actionPerformed(ActionEvent e)
66      {
67          String actionCommand = e.getActionCommand();


68          if (actionCommand.equals("Save Memo 1"))
69              memo1 = memoDisplay.getText();
70          else if (actionCommand.equals("Save Memo 2"))
71              memo2 = memoDisplay.getText();
72          else if (actionCommand.equals("Clear"))
73              memoDisplay.setText("");
74          else if (actionCommand.equals("Get Memo 1"))
75              memoDisplay.setText(memo1);
76          else if (actionCommand.equals("Get Memo 2"))
77              memoDisplay.setText(memo2);
78          else
79              memoDisplay.setText("Error in memo interface");
80       }
81  }
```

### Resulting GUI

9. In Display 18.7, we listed the constructor for `JScrollPane` as follows:

   ```
   public JScrollPane(Component objectToBeScrolled)
   ```

   This indicates that the argument to the constructor must be of type `Component`. But we used the constructor with an argument of type `JTextArea`. Isn't this some sort of type violation?

**Example**

### Components with Changing Visibility

The GUI in Display 18.9 has labels that change from visible to invisible and back again. Since the labels contain nothing but an icon each, it appears as if the icons change roles from visible to invisible and back again. When the GUI is first run, the label with Duke not waving is shown. When the `"Wave"` button is clicked, the label with Duke not waving disappears and the label with Duke waving appears. When the button labeled `"Stop"` is clicked, the label with Duke waving disappears and the label with Duke not waving returns. Note that you can make a component invisible without making the entire GUI invisible.

**Display 18.9 Labels with Changing Visibility *(Part 1 of 3)***

```
1  import javax.swing.JFrame;
2  import javax.swing.ImageIcon;
3  import javax.swing.JPanel;
4  import javax.swing.JLabel;
5  import javax.swing.JButton;
6  import java.awt.Container;
7  import java.awt.BorderLayout;
8  import java.awt.FlowLayout;
9  import java.awt.Color;
10 import java.awt.event.ActionListener;
11 import java.awt.event.ActionEvent;

12 public class VisibilityDemo extends JFrame
13                             implements ActionListener
14 {
15     public static final int WIDTH = 300;
16     public static final int HEIGHT = 200;

17     private JLabel wavingLabel;
18     private JLabel standingLabel;
```

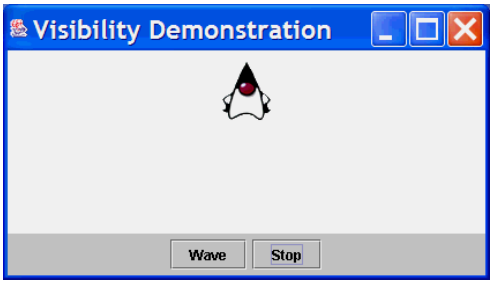
**Display 18.9 Labels with Changing Visibility *(Part 2 of 3)***

```
19      public static void main(String[] args)
20      {
21          VisibilityDemo demoGui = new VisibilityDemo();
22          demoGui.setVisible(true);
23      }

24      public VisibilityDemo()
25      {
26          setSize(WIDTH, HEIGHT);
27          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28          setTitle("Visibility Demonstration");

29          Container contentPane = getContentPane();
30          contentPane.setLayout(new BorderLayout());

31          JPanel picturePanel = new JPanel();
32          picturePanel.setBackground(Color.WHITE);
33          picturePanel.setLayout(new FlowLayout());

34          ImageIcon dukeStandingIcon =
35                    new ImageIcon("duke_standing.gif");
36          standingLabel = new JLabel(dukeStandingIcon);
37          standingLabel.setVisible(true);
38          picturePanel.add(standingLabel);

39          ImageIcon dukeWavingIcon = new ImageIcon("duke_waving.gif");
40          wavingLabel = new JLabel(dukeWavingIcon);
41          wavingLabel.setVisible(false);
42          picturePanel.add(wavingLabel);

43          contentPane.add(picturePanel, BorderLayout.CENTER);

44          JPanel buttonPanel = new JPanel();
45          buttonPanel.setBackground(Color.LIGHT_GRAY);
46          buttonPanel.setLayout(new FlowLayout());

47          JButton waveButton = new JButton("Wave");
48          waveButton.addActionListener(this);
49          buttonPanel.add(waveButton);

50          JButton stopButton = new JButton("Stop");
51          stopButton.addActionListener(this);
52          buttonPanel.add(stopButton);
53
54          contentPane.add(buttonPanel, BorderLayout.SOUTH);
55      }
```

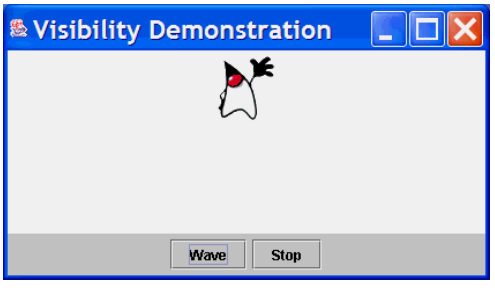
**Display 18.9 Labels with Changing Visibility *(Part 3 of 3)***

```
56      public void actionPerformed(ActionEvent e)
57      {
58          String actionCommand = e.getActionCommand();
59          Container contentPane = getContentPane();

60          if (actionCommand.equals("Wave"))
61          {
62              wavingLabel.setVisible(true);
63              standingLabel.setVisible(false);
64              contentPane.validate();
65          }
66          else if (actionCommand.equals("Stop"))
67          {
68              standingLabel.setVisible(true);
69              wavingLabel.setVisible(false);
70              contentPane.validate();
71          }
72          else
73              System.out.println("Unanticipated error.");
74      }
75  }
```

**RESULTING GUI** (After clicking Stop button)




**RESULTING GUI** (After clicking Wave button)



In this GUI, a label becomes visible or invisible when a button is clicked. For example, the following code from the method `actionPerformed` in Display 18.9 determines what happens when the button with the text "`Wave`" on it is clicked:

```
if (actionCommand.equals("Wave"))
{
    wavingLabel.setVisible(true);
    standingLabel.setVisible(false);
    contentPane.validate();
}
```

The first two statements in the braces make `wavingLabel` visible and `standingLabel` invisible.

We used the `setVisible` method on the panels containing the icons rather than directly on the icons because the class `ImageIcon` does not have the `setVisible` method.

The two statements

```
wavingLabel.setVisible(true);
standingLabel.setVisible(false);
```

make `wavingLabel` visible and `standingLabel` invisible in the representation of the GUI inside the computer. However, making this change show on the screen requires a call to the method `validate`.

validate

Every container class has the method `validate`. An invocation of `validate` causes the container to lay out its components again. An invocation of `validate` is a kind of "update" action that makes changes in the components actually happen on the screen. Many simple changes that are made to a Swing GUI, like changing color or changing the text in a text field, happen automatically. Other changes, such as the addition of components or changes in visibility, often require an invocation of `validate` or some other "update" method. Sometimes it is difficult to decide whether an invocation of `validate` is necessary. When in doubt include an invocation of `validate`. Although invoking `validate` when it is not needed can make your program a little less efficient, it will have no other ill effects on your GUI if you include an extra invocation of `validate`.

We invoked the method `validate` with the content pane. That ensured that everything in the content pane is updated. You always have easy access to the content pane via the method `getContentPane`. Since it is inclusive of most possible changes and since it is easy to name, many programmers use the content pane as the calling object for `validate`. However, you can often use a smaller container. In Display 18.9 all the changes take place in the `JPanel` named `picturePanel`, so we could have instead used `picturePanel` as the calling object for `validate`. This would, however, require that we make `picturePanel` an instance variable so that it could be referenced in both the constructor and the method `actionPerformed`. In the file `VisibilityDemo2.java` on the accompanying CD, we have redone the program in Display 18.9 using `picturePanel` as the calling object for `validate`.

extra code on CD

### The validate Method

Every container class has a method named `validate`, which is a method for updating the container. An invocation of `validate` will cause the container to lay out its components again. As discussed in the text, certain changes to a Swing GUI require an invocation of `validate` to update the screen and make the changes show on the screen.

**Syntax:**

```
Container_Object.validate();
```

The *Container_Object* is often a content pane or an implicit `this`. A complete example of using `validate` can be seen in Display 18.9.

# 18.3 The Graphics Class

*Drawing is my life!*

The `Graphics` class

In this section we show you how to produce drawing for your GUIs using the `Graphics` class.

## COORDINATE SYSTEM FOR GRAPHICS OBJECTS

When drawing objects on the screen, Java uses the coordinate system shown in Display 18.10 The **origin** point (0, 0) is the upper-left corner of the screen area used for drawing (usually a `JFrame` or `JPanel`). The *x*-coordinate, or horizontal coordinate, is positive and increasing to the right. The *y*-coordinate, or vertical coordinate, is positive and increasing in the downward direction. The point (*x*, *y*) is located *x* pixels in from the left edge of the screen and down *y* pixels from the top of the screen. All coordinates are normally positive. Units as well as sizes of figures are in pixels. When placing a rectangle on the screen, Java often uses a coordinate like (200, 150) to specify where the rectangle is located.

origin

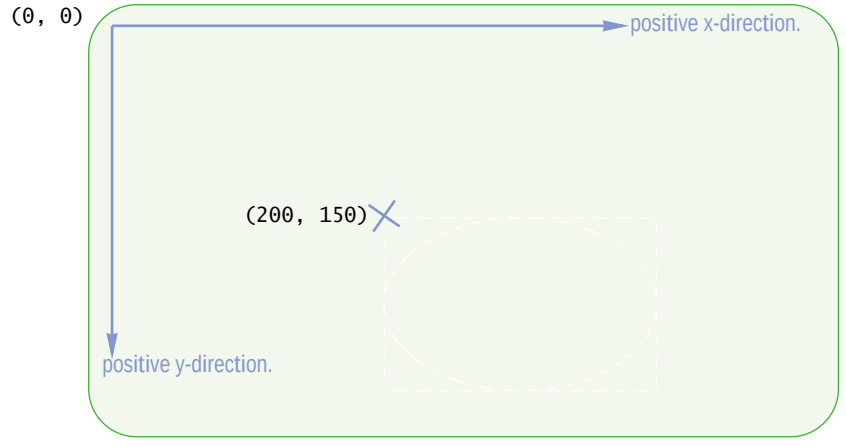
(*x*, *y*)

Note that, when specifying the location of a rectangle or other figure, the coordinates do not indicate the center of the rectangle, but instead indicate the location of the upper-left corner of the rectangle. In Display 18.10, the X marks the location of the point (200, 150) and the rectangle shown is at location (200, 150).

When placing a figure other than a rectangle on the screen, Java encloses the figure in an imaginary tightly fitting rectangle, sometimes called a **bounding box**, and positions the upper-left corner of the imaginary rectangle. For example, in Display 18.10, the oval displayed is located at point (200, 150).

bounding box

**Display 18.10 Screen Coordinate System**




## THE METHOD paint AND THE CLASS Graphics

paint

Almost all Swing and Swing-related components and containers have a method named `paint`. The method `paint` draws the component or container on the screen. Up until now, we have had no need to redefine the method `paint` or to even mention it. The method `paint` is defined for you and is called automatically when the figure is displayed on the screen. However, to draw geometric figures, like circles and boxes, you need to redefine the method `paint`. It is the method `paint` that draws the figures.

Display 18.11 shows a GUI program that displays a `JFrame` with a rather primitive face drawn inside of it. The mouth and eyes are just straight line segments. We will soon see how to get round eyes and a smile (and more), but the basic technique can be seen more clearly in this simple figure. The code for drawing the face is given in the method `paint`.

The method `paint` is called automatically, and you normally should not invoke it in your code. If you do not redefine it, the method `paint` for a `JFrame` object simply draws a frame border, title, and other standard features, and then asks the components to all invoke their `paint` methods. If we did not redefine the method `paint`, then the `JFrame` would have a border and title but would contain nothing. The code in the redefinition of `paint` explains how to draw the face. Let's look at the details.

Graphics

Notice that the method `paint` has a parameter `g` of type `Graphics`. `Graphics` is an abstract class in the `java.awt` package. Every container and component that can be drawn on the screen has an associated `Graphics` object. (To be precise, every `JComponent` has an associated `Graphics` object.) This associated `Graphics` object has data specifying what area of the screen the component or container covers. In particular, the `Graphics` object for a `JFrame` specifies that the drawing takes place inside the borders of the `JFrame`

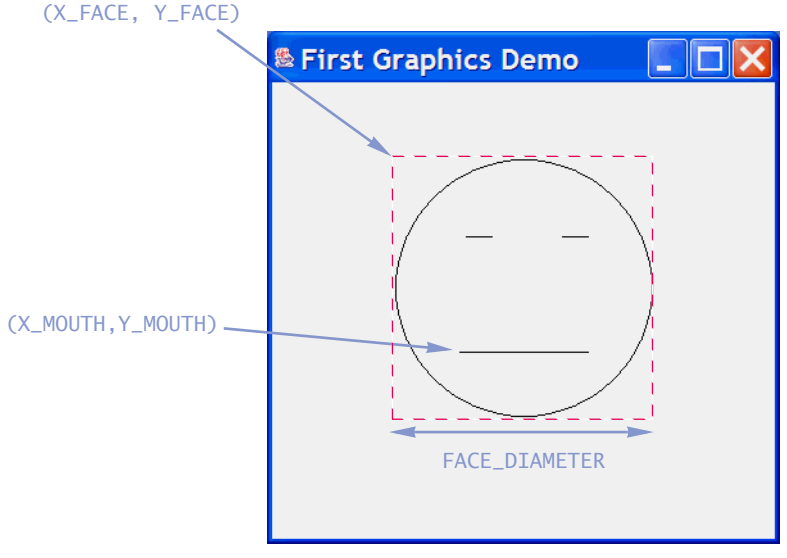
**Display 18.11 Drawing a Very Simple Face *(Part 1 of 2)***

```
1  import javax.swing.JFrame;
2  import java.awt.Graphics;
3  import java.awt.Color;


4  public class Face extends JFrame
5  {
6      public static final int WINDOW_WIDTH = 400;
7      public static final int WINDOW_HEIGHT = 400;


8      public static final int FACE_DIAMETER = 200;
9      public static final int X_FACE = 100;
10     public static final int Y_FACE = 100;


11     public static final int EYE_WIDTH = 20;
12     public static final int X_RIGHT_EYE = X_FACE + 55;
13     public static final int Y_RIGHT_EYE = Y_FACE + 60;
14     public static final int X_LEFT_EYE = X_FACE + 130;
15     public static final int Y_LEFT_EYE = Y_FACE + 60;


16     public static final int MOUTH_WIDTH = 100;
17     public static final int X_MOUTH = X_FACE + 50;
18     public static final int Y_MOUTH = Y_FACE + 150;


19     public static void main(String[] args)
20     {
21         Face drawing = new Face();
22         drawing.setVisible(true);
23     }


24     public Face()
25     {
26         super("First Graphics Demo");
27         setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
28         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29         getContentPane().setBackground(Color.white);
30     }
```

**Display 18.11 Drawing a Very Simple Face *(Part 2 of 2)***

```
31      public void paint(Graphics g)
32      {
33          super.paint(g);
34          g.drawOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
35          //Draw Eyes:
36          g.drawLine(X_RIGHT_EYE, Y_RIGHT_EYE,
37                                   X_RIGHT_EYE + EYE_WIDTH,Y_RIGHT_EYE);
38          g.drawLine(X_LEFT_EYE, Y_LEFT_EYE,
39                                  X_LEFT_EYE + EYE_WIDTH, Y_LEFT_EYE);
40          //Draw Mouth:
41          g.drawLine(X_MOUTH, Y_MOUTH, X_MOUTH + MOUTH_WIDTH, Y_MOUTH);
42      }
43  }
```

**RESULTING GUI**




*The red box is not shown on the screen. It is there to help you understand the relationship between the `paint` method code and the resulting drawing.*

object. (Since `Graphics` is an abstract class, every `Graphics` object is an instance of some concrete descendent class of the `Graphics` class, but we usually do not care about which descendent class. All we normally need to know is that it is of type `Graphics`.)

The `Graphics` class, and so any `Graphics` object `g`, has all the methods that we will use to draw figures, such as circles, lines, and boxes, on the screen. Almost the entire definition of the `paint` method in Display 18.11 consists of invocations of various drawing methods with the parameter `g` as the calling object.

When the `paint` method in Display 18.11 is (automatically) invoked, the parameter `g` will be replaced by the `Graphics` object associated with the `JFrame`, so the figures drawn will be inside the `JFrame`. Let's look at the code in this method `paint`.

Notice the first line in the definition of `paint` in Display 18.11:

```
super.paint(g);
```

Recall that `super` is a name for the parent class of a derived class. The class in Display 18.11 is derived from the class `JFrame`, so `super.paint` is the `paint` method for the class `JFrame`. Whenever you redefine the method `paint`, you should start with this invocation of `super.paint`. This ensures that your definition of `paint` will do all the things the standard `paint` method does, such as draw the title and border for the `JFrame`. (This lesson applies even if the class is derived from some class other than `JFrame`.)

The following invocation from the method `paint` draws the circle forming the head: drawOval

```
g.drawOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
```

The last two arguments give the width and height of the enclosing rectangle, shown in red. The fact that these two arguments are equal is what makes it a circle instead of a typical oval. The first two arguments give *x*- and *y*-coordinates for the position of the circle. Note that a figure is positioned by giving the position of the upper-left corner of an enclosing rectangle.

The only other drawing statements in the method paint are invocations of `g.drawLine`. The method `g.drawLine` draws a straight line between two points with *x*- and *y*-coordinates $(x_1, y_1)$ and $(x_2, y_2)$, where the argument positions for the four coordinate numbers are indicated below: drawLine

```
g.drawLine(x1, y1, x2, y2)
```

For example, the invocation that draws the mouth is as follows:

```
g.drawLine(X_MOUTH, Y_MOUTH, X_MOUTH + MOUTH_WIDTH, Y_MOUTH);
```

Since both *y*-coordinates (`Y_MOUTH`) are the same, the line is horizontal. The line for the mouth begins at coordinates (`X_MOUTH`, `Y_MOUTH`) and extends to the right for `MOUTH_WIDTH` pixels.

Some of the commonly used methods of the class `Graphics` are given in Display 18.12. Note that most methods come in pairs, one whose name starts with `draw` and one whose name starts with `fill`, such as `drawOval` and `fillOval`. The one that starts with `draw` will draw the outline of the specified figure. The one that starts with `fill` will draw a solid figure obtained by filling the inside of the specified figure. In the next few subsections we discuss some of these methods.

### The Graphics Class

Every container and component that can be drawn on the screen has an associated `Graphics` object. This associated `Graphics` object has data specifying what area of the screen the component or container covers. In particular, the `Graphics` object for a `JFrame` specifies that the drawing takes place inside the borders of the `JFrame` object.

When an object g of the class `Graphics` is used as the calling object for a drawing method, the drawing takes place inside the area of the screen specified by g. For example, if g is the `Graphics` object for a `JFrame`, the drawing takes place inside the borders of the `JFrame` object.

Some of the commonly used methods of the class `Graphics` are given in Display 18.12.

`Graphics` is an abstract class in the `java.awt` package.

**Display 18.12 Some Methods in the Class `Graphics` *(Part 1 of 2)***

`Graphics` is an abstract class in the `java.awt` package.

Although many of these methods are abstract, we always use them with objects of a concrete descendent class of `Graphics`, even though we usually do not know the name of that concrete class.

```
public abstract void drawLine(int x1, int y1, int x2, int y2)
```

Draws a line between points (x1, y1) and (x2, y2).

```
public abstract void drawRect(int x, int y,
                              int width, int height)
```

Draws the outline of the specified rectangle. (x, y) is the location of the upper-left corner of the rectangle.

```
public abstract void fillRect(int x, int y,
                              int width, int height)
```

Fills the specified rectangle. (x, y) is the location of the upper-left corner of the rectangle.

```
public void draw3DRect(int x, int y, int width,
                       int height, boolean raised)
```

Draws the outline of the specified rectangle. (x, y) is the location of the upper-left corner. The rectangle is highlighted to look like it has thickness. If `raised` is `true`, the highlight makes the rectangle appear to stand out from the background. If `raised` is `false`, the highlight makes the rectangle appear to be sunken into the background.

```
public void fill3DRect(int x, int y, int width,
                       int height, boolean raised)
```

Fills the rectangle specified by

```
draw3DRec(x, y, width, height, raised)
```

**Display 18.12 Some Methods in the Class Graphics *(Part 2 of 2)***

```
public abstract void drawRoundRect(int x, int y,
            int width, int height, int arcWidth, int arcHeight)
```

Draws the outline of the specified round-cornered rectangle. (x, y) is the location of the upper-left corner of the enclosing regular rectangle. `arcWidth` and `arcHeight` specify the shape of the round corners. See the text for details.

```
public abstract void fillRoundRect(int x, int y,
            int width, int height, int arcWidth, int arcHeight)
```

Fills the rounded rectangle specified by

```
drawRoundRec(x, y, width, height, arcWidth, arcHeight)
```

```
public abstract void drawOval(int x, int y,
                                        int width, int height)
```

Draws the outline of the oval with the smallest enclosing rectangle that has the specified width and height. The (imagined) rectangle has its upper-left corner located at (x, y).

```
public abstract void fillOval(int x, int y,
                                        int width, int height)
```

Fills the oval specified by

```
drawOval(x, y, width, height)
```

```
public abstract void drawArc(int x, int y,
                             int width, int height,
                             int startAngle, int arcSweep)
```

Draws part of an oval that just fits into an invisible rectangle described by the first four arguments. The portion of the oval drawn is given by the last two arguments. See the text for details.

```
public abstract void fillArc(int x, int y,
                             int width, int height,
                             int startAngle, int arcSweep)
```

Fills the partial oval specified by

```
drawArc(x, y, width, height, startAngle, arcSweep)
```

## DRAWING OVALS

An oval is drawn by the method `drawOval`. The arguments specify the location, width, and height of the smallest rectangle that encloses the oval. For example, the following line draws an oval:

drawOval

```
g.drawOval(100, 50, 300, 200);
```

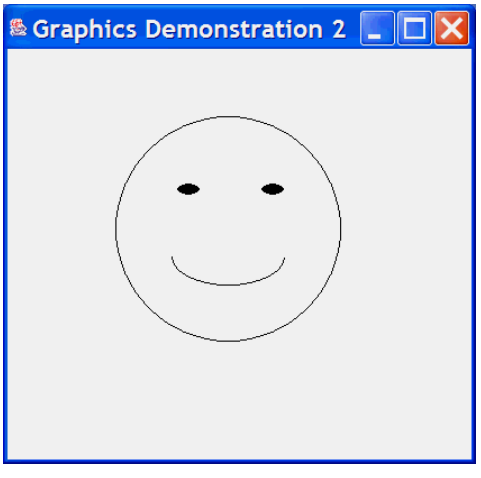
**Display 18.13** **Drawing a Happy Face** ***(Part 1 of 2)***

```
1  import javax.swing.JFrame;
2  import java.awt.Graphics;
3  import java.awt.Color;


4  public class HappyFace extends JFrame
5  {
6      public static final int WINDOW_WIDTH = 400;
7      public static final int WINDOW_HEIGHT = 400;


8      public static final int FACE_DIAMETER = 200;
9      public static final int X_FACE = 100;
10     public static final int Y_FACE = 100;


11     public static final int EYE_WIDTH = 20;
12     public static final int EYE_HEIGHT = 10;
13     public static final int X_RIGHT_EYE = X_FACE + 55;
14     public static final int Y_RIGHT_EYE = Y_FACE + 60;
15     public static final int X_LEFT_EYE = X_FACE + 130;
16     public static final int Y_LEFT_EYE = Y_FACE + 60;


17     public static final int MOUTH_WIDTH = 100;
18     public static final int MOUTH_HEIGHT = 50;
19     public static final int X_MOUTH = X_FACE + 50;
20     public static final int Y_MOUTH = Y_FACE + 100;
21     public static final int MOUTH_START_ANGLE = 180;
22     public static final int MOUTH_ARC_SWEEP = 180;


23     public static void main(String[] args)
24     {
25         HappyFace drawing = new HappyFace();
26         drawing.setVisible(true);
27     }

28     public HappyFace()
29     {
30         super("Graphics Demonstration 2");
31         setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
32         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
33         getContentPane().setBackground(Color.white);
34     }
```

**Display 18.13 Drawing a Happy Face *(Part 2 of 2)***

```
35      public void paint(Graphics g)
36      {
37          super.paint(g);
38          g.drawOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
39          //Draw Eyes:
40          g.fillOval(X_RIGHT_EYE, Y_RIGHT_EYE, EYE_WIDTH, EYE_HEIGHT);
41          g.fillOval(X_LEFT_EYE, Y_LEFT_EYE, EYE_WIDTH, EYE_HEIGHT);
42          //Draw Mouth:
43          g.drawArc(X_MOUTH, Y_MOUTH, MOUTH_WIDTH, MOUTH_HEIGHT,
44                    MOUTH_START_ANGLE, MOUTH_ARC_SWEEP);
45      }
46  }
```

**RESULTING GUI**




This draws an oval that just fits into an invisible rectangle whose upper-left corner is at coordinates (100, 50) and that has a width of 300 pixels and a height of 200 pixels. Note that the point that is used to place the oval on the screen is not the center of the oval or anything like the center, but is something like the upper-left corner of the oval.

Note that a circle is a special case of an oval in which the width and height of the rectangle are equal. For example, the following line from the definition of `paint` in Display 18.11 draws a circle for the outline of the face:

```
g.drawOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
```

Since the enclosing rectangle has the same width and height, this produces a circle.

Some of the methods you can use to draw simple figures are shown in Display 18.12. A similar table is given in Appendix 4.

## DRAWING ARCS

Arcs, such as the smile on the happy face in Display 18.13, are described by giving an oval and then specifying what portion of the oval will be used for the arc. For example, the following statement from Display 18.13 draws the smile on the happy face:

drawArc

```
g.drawArc(X_MOUTH, Y_MOUTH, MOUTH_WIDTH, MOUTH_HEIGHT,
                    MOUTH_START_ANGLE, MOUTH_ARC_SWEEP);
```

which is equivalent to

```
g.drawArc(X_MOUTH, Y_MOUTH, MOUTH_WIDTH, MOUTH_HEIGHT, 180, 180);
```

The arguments `MOUTH_WIDTH` and `MOUTH_HEIGHT` determine the size of an invisible rectangle. The arguments `X_MOUTH` and `Y_MOUTH` determine the location of the rectangle. The upper-left corner of the rectangle is located at the point (`X_MOUTH`, `Y_MOUTH`). Inside this invisible rectangle, envision an invisible oval that just fits inside the invisible rectangle. The last two arguments specify the portion of this invisible oval that is made visible.

Display 18.14 illustrates how these last two arguments specify an arc of the invisible oval to be made visible. The next-to-last argument specifies a start angle in degrees. The last argument specifies how many degrees of the oval's arc will be made visible. If the last argument is 360 (degrees), then the full oval is made visible.

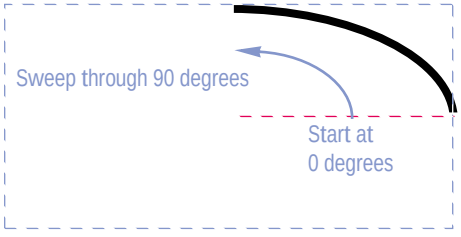
The angles are numbered with zero degrees, as shown in Display 18.14. In the first figure, the start angle is zero degrees. The counterclockwise direction is positive. So a start angle of 90 degrees would start at the top of the oval. A start angle of –90 degrees would start at the bottom of the oval. For example, the smile on the happy face in Display 18.13 has a start angle of 180 degrees, so it starts on the left end of the invisible oval. The last argument is also 180, so the arc is made visible through a counterclockwise direction of 180 degrees, or halfway around the oval in the counterclockwise direction.

### Self-Test Exercises

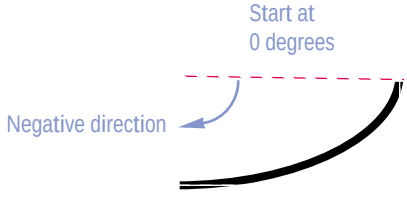
10. Give an invocation of a method to draw a horizontal line from point (30, 40) to point (100, 60). The calling object of type `Graphics` is named `g`.
11. Give an invocation of a method to draw a horizontal line of length 100 starting at position (30, 40) and extending to the right. The calling object of type `Graphics` is named `g`.
12. Give an invocation of a method that draws a vertical line of length 100 starting at position (30, 40) and extending downward. Use `graphicsObject` (of type `Graphic`) as the calling object.

**Display 18.14 Specifying an Arc**

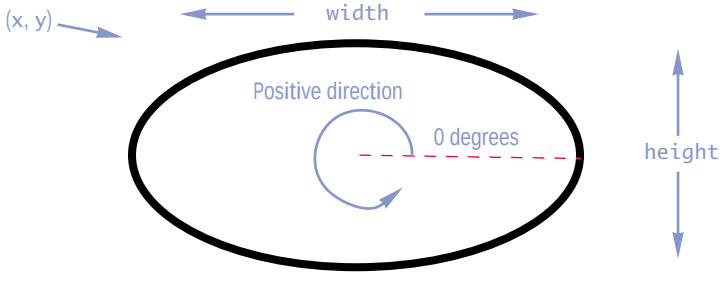
```
g.drawArc(x, y, width, height, 0, 90);
```



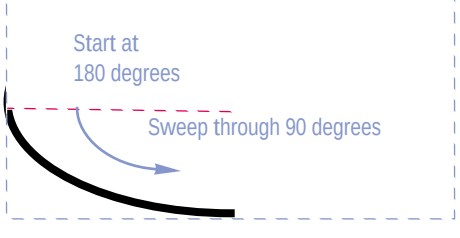

```
g.drawArc(x, y, width, height, 0, –90);
```




```
g.drawArc(x, y, width, height, 0, 360);
```




```
g.drawArc(x, y, width, height, 180, 90);
```



13. Give an invocation of a method to draw a solid rectangle of width 100 and height 50 with the upper-left corner at position (20, 30). The calling object of type `Graphics` is named `graphicsObject`.
14. Give an invocation of a method to draw a solid rectangle of width 100 and height 50 with the upper-right corner at position (200, 300). The calling object of type `Graphics` is named `g`.
15. Give an invocation of a method to draw a circle of diameter 100 with the center at position (300, 400). The calling object of type `Graphics` is named `g`.
16. Give an invocation of a method to draw a circle of radius 100 with the center at position (300, 400). The calling object of type `Graphics` is named `g`.

## ROUNDED RECTANGLES

rounded rectangle

A **rounded rectangle** is a rectangle whose corners have been replaced by arcs so that the corners are rounded. For example, suppose `g` is of type `Graphics` and consider what would be drawn by the following:

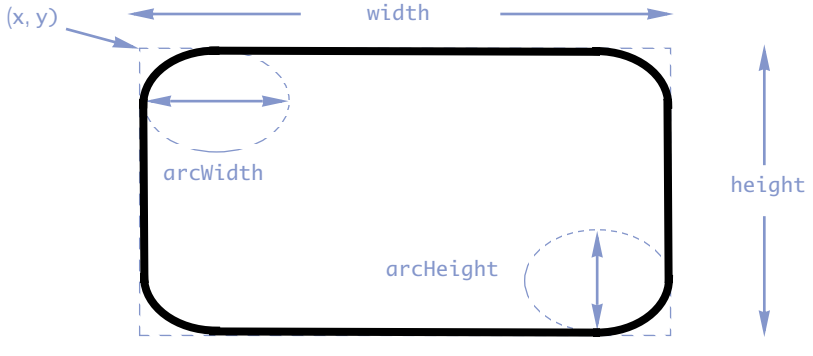
```
g.drawRoundRect(x, y, width, height, arcWidth, arcHeight)
```

The arguments `x`, `y`, `width`, and `height` determine a regular rectangle in the usual way. The upper-left corner is at the point (`x`, `y`). The rectangle has the specified `width` and `height`. The last two arguments, `arcWidth` and `arcHeight`, specify the arcs that will be used for the corners so as to produce a rounded rectangle. Each corner is replaced with a quarter of an oval that is `arcWidth` pixels wide and `arcHeight` pixels high. This is illustrated in Display 18.15. To obtain corners that are arcs of circles just make `arcWidth` and `arcHeight` equal.

**Display 18.15 A Rounded Rectangle**

```
g.drawRoundRect(x, y, width, height, arcWidth, arcHeight);
```

produces:



## paintComponent FOR PANELS

You can draw figures on a `JPanel` and place the `JPanel` in a `JFrame`. When defining a `JPanel` class that contains a graphics drawing, you use the method `paintComponent` instead of the method `paint`, but otherwise the details are similar to what we have seen for `JFrames`. `JFrames` use the method `paint`. However, `JPanels`—and in fact all `JComponents`—use the method `paintComponent`. A very simple example of using `paintComponent` with a `JPanel` is given in Display 18.16.

If you look back at Display 16.12 in Chapter 16, you will see that a `JPanel` is a `JComponent`, but a `JFrame` is not a `JComponent`. A `JFrame` is only a `Component`. That is why they use different methods to paint the screen.

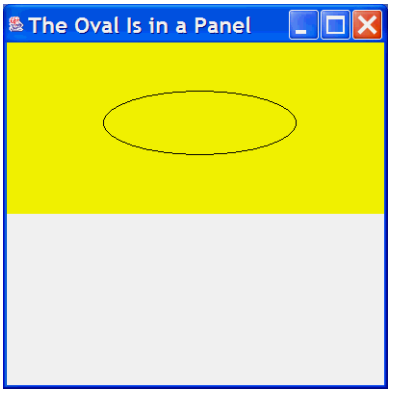
**Display 18.16** **paintComponent Demonstration** ***(Part 1 of 2)***

```
1  import javax.swing.JFrame;
2  import javax.swing.JPanel;
3  import java.awt.GridLayout;
4  import java.awt.Graphics;
5  import java.awt.Container;
6  import java.awt.Color;

7  public class PaintComponentDemo extends JFrame
8  {
9      public static final int FRAME_WIDTH = 400;
10     public static final int FRAME_HEIGHT = 400;

11     private class FancyPanel extends JPanel
12     {
13         public void paintComponent(Graphics g)
14         {
15             super.paintComponent(g);
16             setBackground(Color.YELLOW);
17             g.drawOval(FRAME_WIDTH/4, FRAME_HEIGHT/8,
18                        FRAME_WIDTH/2, FRAME_HEIGHT/6);
19         }
20     }

21     public static void main(String[] args)
22     {
23         PaintComponentDemo w = new PaintComponentDemo();
24         w.setVisible(true);
25     }
```

**Display 18.16 paintComponent Demonstration *(Part 2 of 2)***

```
26      public PaintComponentDemo()
27      {
28          setSize(FRAME_WIDTH, FRAME_HEIGHT);
29          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30          setTitle("The Oval Is in a Panel");
31          Container contentPane = getContentPane();
32          contentPane.setLayout(new GridLayout(2, 1));
33          FancyPanel p = new FancyPanel();
34          contentPane.add(p);
35          JPanel whitePanel = new JPanel();
36          whitePanel.setBackground(Color.WHITE);
37          contentPane.add(whitePanel);
38      }
39  }
```

**RESULTING GUI**




## ACTION DRAWINGS AND repaint

The program in Display 18.17 is similar to the program in Display 18.13. It draws a happy face similar to the happy face given in Display 18.13, but with one difference: There is a button at the bottom of the GUI that says `Click for a Wink`. When you click that button, the left eye winks. (Remember the left eye is on your right.) Let's see the details.

The program in Display 18.17 has a private instance variable `wink` of type `boolean`. When the value of `wink` is `false`, the `paint` method draws an ordinary happy face. When the value of `wink` is `true`, the `paint` method draws the face the same except that the left eye is just a straight line, which looks like the eye is closed. The variable `wink` is initialized to `false`.

**Display 18.17 An Action Drawing *(Part 1 of 3)***

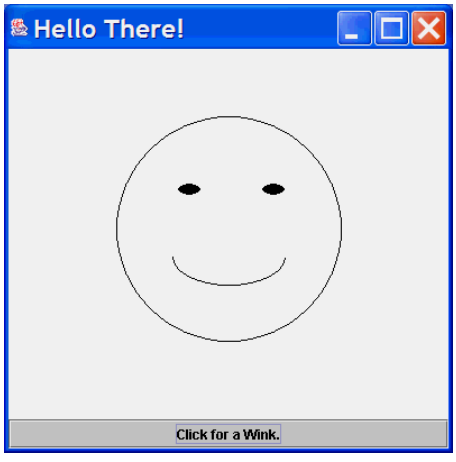
```
1  import javax.swing.JFrame;
2  import javax.swing.JButton;
3  import java.awt.event.ActionListener;
4  import java.awt.event.ActionEvent;
5  import java.awt.Container;
6  import java.awt.BorderLayout;
7  import java.awt.Graphics;
8  import java.awt.Color;

9  public class ActionFace extends JFrame
10 {
11     public static final int WINDOW_WIDTH = 400;
12     public static final int WINDOW_HEIGHT = 400;

13     public static final int FACE_DIAMETER = 200;
14     public static final int X_FACE = 100;
15     public static final int Y_FACE = 100;

16     public static final int EYE_WIDTH = 20;
17     public static final int EYE_HEIGHT = 10;
18     public static final int X_RIGHT_EYE = X_FACE + 55;
19     public static final int Y_RIGHT_EYE = Y_FACE + 60;
20     public static final int X_LEFT_EYE = X_FACE + 130;
21     public static final int Y_LEFT_EYE = Y_FACE + 60;

22     public static final int MOUTH_WIDTH = 100;
23     public static final int MOUTH_HEIGHT = 50;
24     public static final int X_MOUTH = X_FACE + 50;
25     public static final int Y_MOUTH = Y_FACE + 100;
26     public static final int MOUTH_START_ANGLE = 180;
27     public static final int MOUTH_ARC_SWEEP = 180;

28     private boolean wink;

29     private class WinkAction implements ActionListener
30     {
31         public void actionPerformed(ActionEvent e)
32         {
33             wink = true;
34             repaint();
35         }
36     } // End of WinkAction inner class
```

**Display 18.17 An Action Drawing *(Part 2 of 3)***

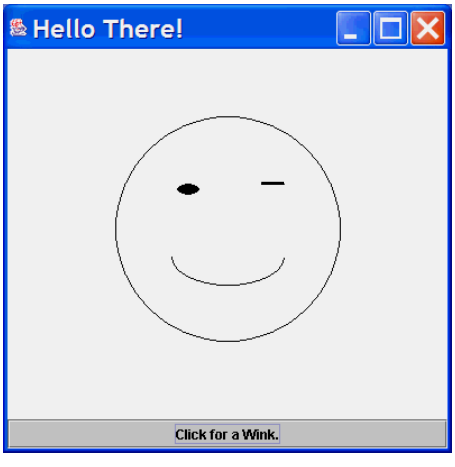
```
37      public static void main(String[] args)
38      {
39          ActionFace drawing = new ActionFace();
40          drawing.setVisible(true);
41      }

42      public ActionFace()
43      {
44          setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
45          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
46          setTitle("Hello There!");
47          Container contentPane = getContentPane();
48          contentPane.setLayout(new BorderLayout());
49          contentPane.setBackground(Color.white);

50          JButton winkButton = new JButton("Click for a Wink.");
51          winkButton.addActionListener(new WinkAction());
52          contentPane.add(winkButton, BorderLayout.SOUTH);
53          wink = false;
54      }

55      public void paint(Graphics g)
56      {
57          super.paint(g);
58          g.drawOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
59          //Draw Right Eye:
60          g.fillOval(X_RIGHT_EYE, Y_RIGHT_EYE, EYE_WIDTH, EYE_HEIGHT);
61          //Draw Left Eye:
62          if (wink)
63              g.drawLine(X_LEFT_EYE, Y_LEFT_EYE,
64                                  X_LEFT_EYE + EYE_WIDTH, Y_LEFT_EYE);
65          else
66              g.fillOval(X_LEFT_EYE, Y_LEFT_EYE, EYE_WIDTH, EYE_HEIGHT);
67          //Draw Mouth:
68          g.drawArc(X_MOUTH, Y_MOUTH, MOUTH_WIDTH, MOUTH_HEIGHT,
69                    MOUTH_START_ANGLE, MOUTH_ARC_SWEEP);
70      }

71  }
```

**Display 18.17 An Action Drawing *(Part 3 of 3)***

**RESULTING GUI** (When started)




**RESULTING GUI** (After clicking the button)




When the button labeled `Click for a Wink` is clicked, this sends an action event to the method `actionPerformed`. The method `actionPerformed` then changes the value of the variable `wink` to `true` and invokes the method `repaint`. This use of the method `repaint` is new, so let's discuss it a bit.

Every `JFrame` (in fact, every `Component` and every `Container`) has a method named `repaint`. The method `repaint` will repaint the screen so that any changes to the graphics

`repaint`

being displayed will show on the screen. If you omit the invocation of `repaint` from the method `actionPerformed`, then the variable `wink` will change to `true`, but the screen will not change. Without an invocation of `repaint`, the face will not change, because the method `paint` must be called again with the new value of `wink` before the change takes effect. The method `repaint` does a few standard things and, most importantly, will also invoke the method `paint`, which redraws the screen. Be sure to note that you should invoke `repaint` and not `paint`.

Now we explain why, when `wink` has the value `true`, the method `paint` draws the face with the left eye changed. The relevant part of the code is the following, which draws the left eye:

```
if (wink)
    g.drawLine(X_LEFT_EYE, Y_LEFT_EYE,
                        X_LEFT_EYE + EYE_WIDTH, Y_LEFT_EYE);
else
    g.fillOval(X_LEFT_EYE, Y_LEFT_EYE, EYE_WIDTH, EYE_HEIGHT);
```

If `wink` has the value `true`, then the eye is drawn as a line, which looks like a closed eye. If `wink` has the value `false`, then the eye is drawn as an oval, which looks like an open eye.

### The `repaint` and `paint` Methods

When you change the graphic's contents in a window and want to update the window so that the new contents show on the screen, do not call `paint`; call `repaint`. The `repaint` method takes care of some overhead and then calls the `paint` method. Normally, you do not define `repaint`. As long as you define the `paint` method correctly, the `repaint` method should work correctly. Note that you often define `paint`, but you normally do not call `paint`. On the other hand, normally you do not define `repaint`, but you do sometimes call `repaint`.

## SOME MORE DETAILS ON UPDATING A GUI

repaint manager

With Swing, most changes to a GUI windowing system that we have seen are updated automatically so that they are visible on the screen. This is done by an object known as the **repaint manager**. The repaint manager works automatically, and you need not even be aware of its presence. However, there are a few updates that the repaint manager will not do for you. You have already learned that you need an invocation of `validate` when you change the components in a container or change their visibility as in Display 18.9. You also need to update the screen, this time with an invocation of the method `repaint`, when your GUI changes the figure drawn in the `JFrame` as in Display 18.17.

pack

One other updating method that you will often see when looking at Swing code is `pack`. The method `pack` causes the window to be resized, usually to a smaller size, but

more precisely to an approximation of a size known as the preferred size. (Yes, you can change the preferred size, but we do not have room to cover all of the Swing library in these few chapters.)

We do not have room in this book to go into all the details of how a GUI is updated on the screen, but these few remarks may make some code you find in more advanced books a little less puzzling.

# 18.4 Colors

*One colored picture is worth a thousand black and white pictures.*

Variation on a Chinese proverb

In this section we tell you how to specify colors for the figures you draw with the graphics methods. We also show you how to define your own colors using the class `Color`.

## SPECIFYING A DRAWING COLOR

When drawing figures with methods such as `drawLine` inside of the definition of the `paint` method, you can think of your drawing as being done with a pen that can change colors. The method `setColor` will change the color of the pen.

setColor

For example, consider the happy face that is drawn by the GUI in Display 18.13. If you change the definition of the `paint` method to the version shown in Display 18.18, the eyes will be blue and the mouth will be red. (The file `HappyFaceColor.java` on the accompanying CD contains a version of the changed program. It consists of the program in Display 18.13 with the definition of the `paint` method replaced by the one in Display 18.13 and with the class name changed from `HappyFace` to `HappyFaceColor`.)

extra code on CD

### THE setColor METHOD

When you are doing drawings with an object of the class `Graphics`, you can set the color of the drawing with an invocation of `setColor`. The color specified can later be changed with another invocation of `setColor`, so a single drawing can have multiple colors.

**SYNTAX:**

```
Graphics_Object.setColor(Color_Object);
```

**EXAMPLE:**

```
g.setColor(Color.BLUE);
```

**Display 18.18 Adding Color**

```
1       public void paint(Graphics g)
2       {
3           super.paint(g);
4           //Default is equivalent to: g.setColor(Color.black);
5           g.drawOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
6           //Draw Eyes:
7           g.setColor(Color.BLUE);
8           g.fillOval(X_RIGHT_EYE, Y_RIGHT_EYE, EYE_WIDTH, EYE_HEIGHT);
9           g.fillOval(X_LEFT_EYE, Y_LEFT_EYE, EYE_WIDTH, EYE_HEIGHT);
10          //Draw Mouth:
11          g.setColor(Color.RED);
12          g.drawArc(X_MOUTH, Y_MOUTH, MOUTH_WIDTH, MOUTH_HEIGHT,
13                    MOUTH_START_ANGLE, MOUTH_ARC_SWEEP);
14      }
```

*If you replace the method `paint` in Display 18.13 with this version of `paint`, then the happy face will have blue eyes and red lips.*

## DEFINING COLORS

Display 16.5 in Chapter 16 lists the standard colors in the class `Color`, which are defined for you. If that table does not have the colors you want, you can use the class `Color` to define your own colors. To understand how this is done, you need to first know a few basic facts about colors. By mixing red, green, and blue light in varying amounts, the human eye can be given the sensation of seeing any color the eye is capable of seeing. This is what an ordinary television set does to produce all the colors it displays. The television mixes red, green, and blue light and shines these lights on the screen in differing amounts. This is often called the **RGB color system**, for obvious reasons. Since a computer monitor is basically the same thing as a television set, colors for computer monitors can be produced in the same way. The Java `Color` class mixes amounts of red, green, and blue to produce any new color you might want.

RGB color system

When specifying the amount of each of the colors red, green, and blue, you can use either integers in the range `0` to `255` (inclusive) or `float` values in the range `0.0` to `1.0` (inclusive). For example, brown is formed by mixing red and green. So, the following defines a color called `brown` that will look like a shade of brown:

`Color` constructors

```
Color brown = new Color(200, 150, 0);
```

This color `brown` will have a `200.0/255` fraction of the maximum amount of red possible, a `150.0/255` fraction of the maximum amount of green possible, and no blue. If you want to use fractions to express the color, you can. The following is an equivalent way of defining the same color `brown`:

```
Color brown =
new Color((float)(200.0/255), (float)(150.0/255), (float)0.0);
```

You need the type casts `(float)` because the constructors for the class `Color` only accept arguments of type `int` or `float`, and numbers like `200.0/255` and `0.0` are considered to be of type `double`, not of type `float`.

Some constructors for the class `Color` and some of the commonly used methods for the class `Color` are summarized in Display 18.19.

### RGB Colors

The class `Color` uses the RGB method of creating colors. That means that every color is a combination of the three colors red, green, and blue.

**Pitfall**

### Using doubles to Define a Color

Suppose you want to make a color that is made of half the possible amount of red, half the possible amount of blue, and no green. The following seems reasonable:

```
Color purple = new Color(0.5, 0.0, 0.5);
```

However, this will produce a compiler error. The numbers `0.5` and `0.0` are considered to be of type `double`, and this constructor requires arguments of type `float` (or of type `int`). So, an explicit type cast is required, as follows:

```
Color purple = new Color((float)0.5, (float)0.0, (float)0.5);
```

Java does allow the following method of specifying that a number is of type `float`, and this can be simpler than the previous line of code:

```
Color purple = new Color(0.5f, 0.0f, 0.5f);
```

An even easier way to avoid these problems is to simply use `int` arguments, as in the following:

```
Color purple = new Color(127, 0, 127);
```

(You may feel that the values of 127 should be replaced by 128, but that is a minor point. You are not likely to even notice the difference in color between, say, 127 red and 128 red.)

In any final code produced, these `float` numbers should normally be replaced by defined constants, such as

```
public static final float RED_VALUE = (float)0.5;
public static final float GREEN_VALUE = (float)0.0;
public static final float BLUE_VALUE = (float)0.5;
```

Note that even though the defined constants are specified to be of type `float`, you still need a type cast.

**Display 18.19 Some Methods in the Class Color**

The class `Color` is in the `java.awt` package.

`public Color(int r, int g, int b)`

Constructor that creates a new `Color` with the specified RGB values. The parameters `r`, `g`, and `b` must each be in the range `0` to `255` (inclusive).

`public Color(float r, float g, float b)`

Constructor that creates a new `Color` with the specified RGB values. The parameters `r`, `g`, and `b` must each be in the range `0.0` to `1.0` (inclusive).

`public int getRed()`

Returns the red component of the calling object. The returned value is in the range `0` to `255` (inclusive).

`public int getGreen()`

Returns the green component of the calling object. The returned value is in the range `0` to `255` (inclusive).

`public int getBlue()`

Returns the blue component of the calling object. The returned value is in the range `0` to `255` (inclusive).

`public Color brighter()`

Returns a brighter version of the calling object color.

`public Color darker()`

Returns a darker version of the calling object color.

`public boolean equals(Object c)`

Returns `true` if `c` is equal to the calling object color; otherwise, returns `false`.

## THE JColorChooser DIALOG WINDOW

The class `JColorChooser` can be used to produce a dialog window that allows you to choose a color by looking at color samples or by choosing RGB values. The static method `showDialog` in the class `JColorChooser` produces a window that allows the user to choose a color. A sample program using this method is given in Display 18.20. The statement that launches the `JColorChooser` dialog window is the following:

```
sampleColor =
   JColorChooser.showDialog(this, "JColorChooser", sampleColor);
```

When this statement is executed, the window shown in the second GUI picture in Display 18.20 is displayed for the user to choose a color. Once the user has chosen a color and clicked the `OK` button, the window goes away and the chosen color is returned as

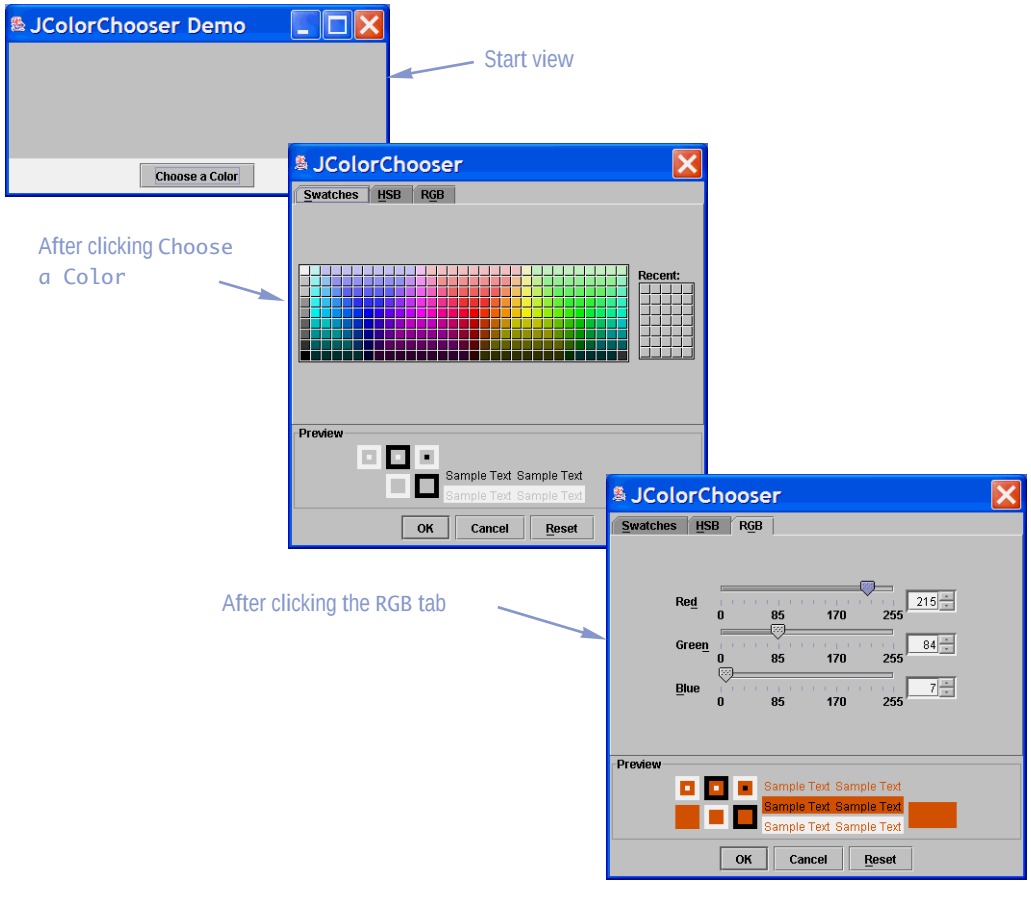
**Display 18.20 JColorChooser Dialog *(Part 1 of 2)***

```
1  import javax.swing.JFrame;
2  import javax.swing.JPanel;
3  import javax.swing.JButton;
4  import javax.swing.JColorChooser;
5  import java.awt.event.ActionListener;
6  import java.awt.event.ActionEvent;
7  import java.awt.Container;
8  import java.awt.BorderLayout;
9  import java.awt.FlowLayout;
10 import java.awt.Color;


11 public class JColorChooserDemo extends JFrame
12                                 implements ActionListener
13 {
14     public static final int WIDTH = 400;
15     public static final int HEIGHT = 200;

16     private Container contentPane;
17     private Color sampleColor = Color.LIGHT_GRAY;


18     public static void main(String[] args)
19     {
20         JColorChooserDemo gui = new JColorChooserDemo();
21         gui.setVisible(true);
22     }

23     public JColorChooserDemo()
24     {
25         contentPane = getContentPane();
26         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27         contentPane.setBackground(sampleColor);
28         contentPane.setLayout(new BorderLayout());
29         setTitle("JColorChooser Demo");
30         setSize(WIDTH, HEIGHT);
31         JPanel buttonPanel = new JPanel();
32         buttonPanel.setBackground(Color.WHITE);
33         buttonPanel.setLayout(new FlowLayout());
34         JButton chooseButton = new JButton("Choose a Color");
35         chooseButton.addActionListener(this);
36         buttonPanel.add(chooseButton);
37         contentPane.add(buttonPanel, BorderLayout.SOUTH);
38     }
```

**Display 18.20 JColorChooser Dialog *(Part 2 of 2)***

```
39      public void actionPerformed(ActionEvent e)
40      {
41          if (e.getActionCommand().equals("Choose a Color"))
42          {
43              sampleColor =
44                 JColorChooser.showDialog(this, "JColorChooser", sampleColor);
45              if (sampleColor != null)//If a color was chosen
46                  contentPane.setBackground(sampleColor);
47          }
48          else
49              System.out.println("Unanticipated Error");
50      }
51  }
```

**RESULTING GUI** (Three views of one GUI)



the value of the JColorChooser.showDialog method invocation. So, in this example, the Color object returned is assigned to the variable sampleColor. If the user clicks the Cancel button, then the method invocation returns null rather than a color.

The method JColorChooser.showDialog takes three arguments. The first argument is the parent component, which is the component from which it was launched. In most simple cases, it is likely to be this, as it is in our example. The second argument is a title for the color chooser window. The third argument is the initial color for the color chooser window. The window shows the user samples of what the color he or she chooses will look like. The user can choose colors repeatedly, and each will be displayed in turn until the user clicks the OK button. The color displayed when the color chooser window first appears is that third argument.

The color chooser window has three tabs at the top labeled Swatches, HSB, and RGB. This gives the user three different ways to choose colors. If the Swatches tab is clicked, the window displays color samples for the user to choose from. This is the way the window first comes up. So, if the user clicks no tab, it is the same as clicking the Swatches tab. The RGB tab allows the user to choose a color by specifying the red, green, and blue values. The HSB tab gives the user a chance to choose colors in a way we will not discuss. To really understand the JColorChooser dialog window, you need to run the program in Display 18.20 to see it in action.

### Self-Test Exercises

17. How would you change the method paint in Display 18.18 so that the happy face has one blue eye (the right eye) and one green eye (the left eye)?
18. How would you change the method paint in Display 18.18 so that the happy face not only has blue eyes and a red mouth, but also has brown skin?

## 18.5 Fonts and the drawString Method

*It is not of so much consequence what you say,*
*as how you say it.*

Alexander Smith, *Dreamthorp. On the Writing of Essays*

Java has facilities to add text to drawings and to modify the font of the text. We will show you enough to allow you to do most things you might want to do with text and fonts.

### THE drawString METHOD

Display 18.21 contains a demonstration program for the method drawString. When the program is run, the GUI displays the text "Push the button.". When the user

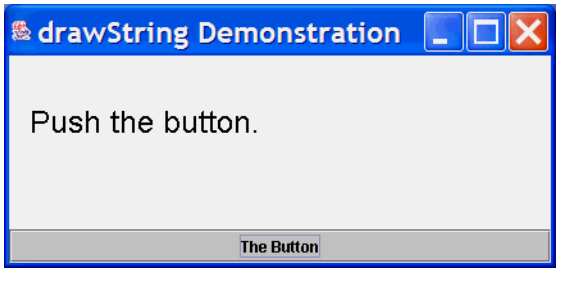
**Display 18.21 Using drawString *(Part 1 of 3)***

```
1  import javax.swing.JFrame;
2  import javax.swing.JPanel;
3  import javax.swing.JButton;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import java.awt.Container;
7  import java.awt.BorderLayout;
8  import java.awt.Graphics;
9  import java.awt.Color;
10 import java.awt.Font;

11 public class DrawStringDemo extends JFrame
12                             implements ActionListener
13 {
14     public static final int WIDTH = 350;
15     public static final int HEIGHT = 200;
16     public static final int X_START = 20;
17     public static final int Y_START = 100;
18     public static final int POINT_SIZE = 24;

19     private String theText = "Push the button.";
20     private Color penColor = Color.BLACK;
21     private Font fontObject =
22                     new Font("SansSerif", Font.PLAIN, POINT_SIZE);

23     public static void main(String[] args)
24     {
25         DrawStringDemo gui = new DrawStringDemo();
26         gui.setVisible(true);
27     }


28     public DrawStringDemo()
29     {
30         setSize(WIDTH, HEIGHT);
31         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
32         setTitle("drawString Demonstration");

33         Container contentPane = getContentPane();
34         contentPane.setBackground(Color.WHITE);
35         contentPane.setLayout(new BorderLayout());

36         JPanel buttonPanel = new JPanel();
37         buttonPanel.setBackground(Color.ORANGE);
38         buttonPanel.setLayout(new BorderLayout());
```

**Display 18.21 Using drawString *(Part 2 of 3)***

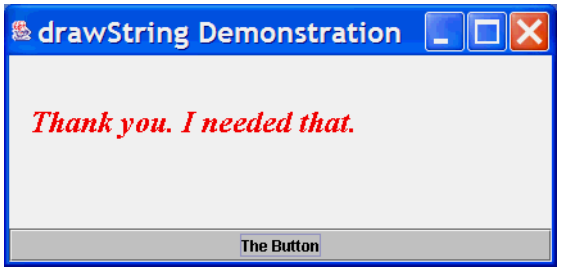
```
39          JButton theButton = new JButton("The Button");
40          theButton.addActionListener(this);

41          buttonPanel.add(theButton, BorderLayout.CENTER);

42          contentPane.add(buttonPanel, BorderLayout.SOUTH);
43      }

44      public void paint(Graphics g)
45      {
46          super.paint(g);
47          g.setFont(fontObject);
48          g.setColor(penColor);
49          g.drawString(theText, X_START, Y_START);
50      }

51      public void actionPerformed(ActionEvent e)
52      {
53          penColor = Color.RED;
54          fontObject =
55                 new Font("Serif", Font.BOLD|Font.ITALIC, POINT_SIZE);
56          theText = "Thank you. I needed that.";

57          repaint();
58      }
59  }
```

**RESULTING GUI** (Start view)



**Display 18.21 Using drawString *(Part 3 of 3)***

**RESULTING GUI** (After clicking the button)




clicks the button, the string is changed to `"Thank you. I needed that."`. The text is written with the method `drawString`.

The method `drawString` is similar to the drawing methods in the class `Graphics`, but it displays text rather than a drawing. For example, the following line from Display 18.21 writes the string stored in the variable `theText` starting at the *x*- and *y*-coordinates `X_START` and `Y_START`:

```
g.drawString(theText, X_START, Y_START);
```

The string is written in the current font. A default font is used if no font is specified. The details about fonts are discussed in the next subsection.

## FONTS

The program in Display 18.21 illustrates how the font for the method `drawString` is set. That program sets the font with the following line in the definition of the method `paint`:

setFont

```
g.setFont(fontObject);
```

In that program `fontObject` is a private instance variable of type `Font`. `Font` is a class in the `java.awt` package. Objects of the class `Font` represent fonts.

In Display 18.21 the variable `fontObject` is set using a constructor for the class `Font`. The initial font is set as part of the instance variable declaration in the following lines taken from Display 18.21:

```
private Font fontObject =
                    new Font("SansSerif", Font.PLAIN, POINT_SIZE);
```

Font

The constructor for the class `Font` creates a font in a given style and size. The first argument, in this case `"SansSerif"`, is a string that gives the name of the font (that is, the

basic style). Some typical font names are `"Times"`, `"Courier"`, and `"Helvetica"`. You may use any font currently available on your system. Java guarantees that you will have at least the three fonts `"Monospaced"`, `"SansSerif"`, and `"Serif"`. To see what these fonts look like on your system, run the program `FontDisplay.java` on the accompanying CD. It will produce the window shown in Display 18.22.

extra code on CD

Most font names have no real meaning. The names just sounded right to the creator. However, the terms "Serif," "Sans Serif," and "Monospaced" do mean something, which may help you keep the names of the three guaranteed fonts clear in your mind. **Serifs** are those small lines that sometimes finish off the ends of the lines in letters. For example, S has serifs (at the two ends of the curved line), but `S` does not have serifs. The `"Serif"` font will always have these decorative little lines. *Sans* means without, so the `"SansSerif"` font will not have these decorative little lines. As you might guess, "`Monospaced`" means that all the characters have equal width.

Fonts can be given style modifiers, such as bold or italic, and they can come in different sizes. The second and third arguments to the constructor for `Font` specify the style modifications and size for the font, as in the following, which occurs in the `actionPerformed` method in Display 18.21:

```
new Font("Serif", Font.BOLD|Font.ITALIC, POINT_SIZE);
```

The second argument specifies style modifications. Note that you can specify multiple style modifications by connecting them with the symbol `|` as in `Font.BOLD|Font.ITALIC`.[2] The last argument specifies the size of the letters in the version of the font created.

**Display 18.22** **Result of Running `FontDisplay.java`**

*Fonts may look somewhat different on your system.*

point size

Character sizes are specified in units known as *points*, so the size of a particular version of a font is called a **point size**. One **point** is 1/72 of an inch, but measurements of font sizes are not as precise as might be ideal; two different fonts of the same point size may be slightly different in size.

The method `setFont` sets the font for the `Graphics` object, which is named `g` in Display 18.21. The font remains in effect until it is changed. If you do not specify any font, then a default font is used.

There is no simple way to change the properties of the current font, such as making it italic. Every change in a font normally requires that you define a new `Font` object and use it as an argument to `setFont`.

Display 18.23 gives some useful details about constructors, methods, and constants that are members of, or are related to, the class `Font`.

### The drawString Method

The `drawString` method writes the text given by the *String* at the point (*X*, *Y*) of the *Graphics_Object*. The text is written in the current font, color, and font size.

**Syntax:**

```
Graphics_Object.drawString(String, X, Y);
```

**Example:**

```
g.drawString("I love you madly.", X_START, Y_START);
```

**Display 18.23 Some Methods and Constants for the Class Font *(Part 1 of 2)***

The class `Font` is in the `java.awt` package.

**CONSTRUCTOR FOR THE CLASS Font**

```
public Font(String fontName, int styleModifications, int size)
```

Constructor that creates a version of the font named by `fontName` with the specified `styleModifications` and `size`.

[2] The symbol | produces a "bitwise or of the numbers," but that detail need not concern you. You need not even know what is meant by a "bitwise or of the numbers." Just think of | as a special way to connect style specifications.

**Display 18.23 Some Methods and Constants for the Class Font *(Part 2 of 2)***

| CONSTANTS IN THE CLASS Font |
|---|
| `Font.BOLD`<br>Specifies bold style. |
| `Font.ITALIC`<br>Specifies italic style. |
| `Font.PLAIN`<br>Specifies plain style—that is, not bold and not italic. |
| **NAMES OF FONTS (These three are guaranteed by Java. Your system will probably have others as well as these.)** |
| `"Monospaced"`<br>See Display 18.22 for a sample. |
| `"SansSerif"`<br>See Display 18.22 for a sample. |
| `"Serif"`<br>See Display 18.22 for a sample. |
| **METHOD THAT USES Font** |
| `public abstract void setFont(Font fontObject)`<br>This method is in the class `Graphics`. Sets the current font of the calling `Graphics` object to `fontObject`. |

## Self-Test Exercises

19. Suppose `g` is an object of type `Graphics`. Write a line of code that will set the font for `g` to Sans Serif bold of size 14 points.
20. Suppose `g` is an object of type `Graphics`. Write a line of code that will set the font for `g` to Sans Serif bold and italic of size 14 points.

## Chapter Summary

- You can define a window listener class by having it implement the `WindowListener` interface.
- An icon is an object of the class `ImageIcon` and is created from a digital picture. You can add icons to `JButton`s, `JLabel`s, and `JMenuItem`s.

- You can use the class `JScrollPane` to add scroll bars to a text area.
- You can draw figures such as lines, ovals, and rectangles using methods in the class `Graphics`.
- You can use the method `setColor` to specify the color of each figure or text drawn with the method of the class `Graphics`.
- You can define your own colors using the class `Color`.
- Colors are defined using the RGB (red/green/blue) system.
- You can use the method `drawString` of the class `Graphics` to add text to a `JFrame` or `JPanel`.
- You can use the method `setFont` to set the font, style modifiers, and point size for text written with the `drawString` method of the `Graphics` class.

## Answers to Self-Test Exercises

1. All the methods in Display 18.1. If there is no particular action that you want the method to perform, you can give the method an empty body.
2. The smaller window goes away but the larger window stays. This is the default action for the close-window button and we did not change it for the smaller window.
3. `dispose`
4. The import statements are the same as in Display 18.2. The rest of the definition follows. This definition is in the file `WindowListenerDemo3` on the accompanying CD.

extra code on CD

```
public class WindowListenerDemo3 extends JFrame
                                  implements WindowListener
{
    public static final int WIDTH = 300; //for main window
    public static final int HEIGHT = 200; //for main window
    public static final int SMALL_WIDTH = 200; //for confirm window
    public static final int SMALL_HEIGHT = 100;//for confirm window

    private class ConfirmWindow extends JFrame
                                implements ActionListener
    {
        public ConfirmWindow()
        {
            setSize(SMALL_WIDTH, SMALL_HEIGHT);
            Container confirmContent = getContentPane();
            confirmContent.setBackground(Color.YELLOW);
            confirmContent.setLayout(new BorderLayout());

            JLabel confirmLabel = new JLabel(
                          "Are you sure you want to exit?");
            confirmContent.add(confirmLabel, BorderLayout.CENTER);
```

```
        JPanel buttonPanel = new JPanel();
        buttonPanel.setBackground(Color.ORANGE);
        buttonPanel.setLayout(new FlowLayout());

        JButton exitButton = new JButton("Yes");
        exitButton.addActionListener(this);
        buttonPanel.add(exitButton);

        JButton cancelButton = new JButton("No");
        cancelButton.addActionListener(this);
        buttonPanel.add(cancelButton);

        confirmContent.add(buttonPanel, BorderLayout.SOUTH);
    }

    public void actionPerformed(ActionEvent e)
    {
        String actionCommand = e.getActionCommand();

        if (actionCommand.equals("Yes"))
            System.exit(0);
        else if (actionCommand.equals("No"))
            dispose();//Destroys only the ConfirmWindow.
        else
            System.out.println(
                   "Unexpected Error in Confirm Window.");
    }
} //End of inner class ConfirmWindow

public static void main(String[] args)
{
    WindowListenerDemo3 demoWindow =
                          new WindowListenerDemo3();
    demoWindow.setVisible(true);
}

public WindowListenerDemo3()
{
    setSize(WIDTH, HEIGHT);
    setTitle("Window Listener Demonstration");

    setDefaultCloseOperation(
                JFrame.DO_NOTHING_ON_CLOSE);
    addWindowListener(this);

    Container contentPane = getContentPane();
    contentPane.setBackground(Color.LIGHT_GRAY);
```

```
        JLabel aLabel =
                  new JLabel("I like to be sure you are sincere.");
        contentPane.add(aLabel);
    }

    //The following are now methods of the class WindowListenerDemo3:
    public void windowOpened(WindowEvent e)
    {}

    public void windowClosing(WindowEvent e)
    {
        ConfirmWindow checkers = new ConfirmWindow();
        checkers.setVisible(true);
    }

    public void windowClosed(WindowEvent e)
    {}

    public void windowIconified(WindowEvent e)
    {}

    public void windowDeiconified(WindowEvent e)
    {}

    public void windowActivated(WindowEvent e)
    {}

    public void windowDeactivated(WindowEvent e)
    {}
}
```

5. 
```
JButton magicButton = new JButton("Magic Button");
ImageIcon wizardIcon = new ImageIcon("wizard.gif");
magicButton.setIcon(wizardIcon);
```

   There are a number of other ways to accomplish the same thing. Below are two of a number of valid alternatives:

```
JButton magicButton = new JButton("Magic Button");
magicButton.setIcon(new ImageIcon("wizard.gif"));

ImageIcon wizardIcon = new ImageIcon("wizard.gif");
JButton magicButton =
                new JButton("Magic Button", wizardIcon);
```

6. 
```
ImageIcon wizardIcon = new ImageIcon("wizard.gif");
JLabel wizardPicture = new JLabel(wizardIcon);
picturePanel.add(wizardPicture);
```

There are a number of other ways to accomplish the same thing. Below is one valid alternative:

```
picturePanel.add(new JLabel(
                     new ImageIcon("wizard.gif")));
```

7. 
```
ImageIcon wizardIcon = new ImageIcon("wizard.gif");
JButton magicButton = new JButton(wizardIcon);
magicButton.setActionCommand("Kazam");
```

   There are a number of other ways to accomplish the same thing. Below is one valid alternative:

```
JButton magicButton =
           new JButton(new ImageIcon("wizard.gif"));
magicButton.setActionCommand("Kazam");
```

8. No. You can invoke none, one, or both methods.
9. No. The class `JTextArea` is a descendent class of the class `Component`. So, every `JTextArea` is also a `Component`.
10. `g.drawLine(30, 40, 100, 60);`
11. `g.drawLine(30, 40, 130, 40);`
12. `graphicsObject.drawLine(30, 40, 30, 140);`
13. `graphicsObject.fillRect(20, 30, 100, 50);`
14. `g.fillRect(100, 300, 100, 50);`
15. `g.drawOval(250, 350, 100, 100);`
16. `g.drawOval(200, 300, 200, 200);`
17. Insert `g.setColor(Color.GREEN)` as indicated below:

```
//Draw Eyes:
g.setColor(Color.BLUE);
g.fillOval(X_RIGHT_EYE, Y_RIGHT_EYE, EYE_WIDTH, EYE_HEIGHT);
g.setColor(Color.GREEN);
g.fillOval(X_LEFT_EYE, Y_LEFT_EYE, EYE_WIDTH, EYE_HEIGHT);
```

18. Replace the following line in the `paint` method:

```
g.drawOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
```

    with

```
Color brown =
        new Color(200, 150, 0);
g.setColor(brown);
g.fillOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
```

Note that there is no predefined color constant `Color.BROWN`, so you need to define a color for brown. You may prefer some other arguments instead of `(200, 150, 0)` so that you get a shade of brown that is more to your liking.

19. `g.setFont(new Font("SansSerif", Font.BOLD, 14));`
20. 
```
g.setFont(new Font("SansSerif",
                                Font.BOLD|Font.ITALIC, 14));
```

## Programming Projects

1. Write a "skeleton" GUI program that implements the `WindowListener` interface. Write code for each of the methods in Display 18.1 that simply prints out a message identifying which event occurred. Print the message out in a text field. Note that your program will not end when the close-window button is clicked (but will instead simply send a message to the text field saying that the `windowClosing` method has been invoked). Include a button labeled `Exit` that the user can click to end the program.
2. Enhance the face drawing in Display 18.17 in all of the following ways: Add color so the eyes are blue and the mouth is red. When the face winks, the line that represents a closed eye is black not blue. Add a nose and a brown handlebar mustache. Add buttons labeled `"Smile"` and `"Frown"`. When the `"Frown"` button is clicked, the face shows a frown (upside down smile); when the `"Smile"` button is clicked, the face shows a smile. When the user clicks the close-window button, a window pops up to ask if the user is sure he or she wants to exit, as in Display 18.2.
3. Write a GUI program to sample different fonts. The user enters a line of text in a text field. The user then selects a font from a font menu. Offer the three guaranteed fonts and at least two other fonts. The user also selects any desired style modifiers (bold and/or italic) from a style menu, and selects point size from a point size menu that offers the following choices: 9, 10, 12, 14, 16, 24, and 32. There is also a `"Display"` button. When the `"Display"` button is clicked, the text is displayed in the font, style, and point size chosen.