

## Getting Started

In this section, we'll see Java in action. Some demo programs will highlight key features of Java. Since we're just getting started, these programs are intended only to pique your interest. We'll worry about the details later. Our goal is to see Java in action, to get a perspective on the journey that lies ahead.

## Software Tools

The following four software tools are needed to learn to program in Java:

An editor	You'll need an editor to enter and save Java programs. If you're working from a <i>DOS</i> window, you can use <code>EDIT</code> or <code>Notepad</code> . If you're working in a unix environment (e.g., <i>linus</i> , <i>Solaris</i> ), you can use <code>vi</code> or <code>pico</code> . However, we'll leave the choice of editor to you. The file you create with an editor contains the <i>source code</i> for a Java program. The file must have ". java" appended to the filename.
javac	<code>javac</code> is the name of the Java compiler. It is the program that translates a Java source code program into Java <i>byte codes</i> . The file created by <code>javac</code> has ".class" as the filename suffix. The information in the ".class" file is often called <i>object code</i> to distinguish it from the source code in the ". java" file.
java	<code>java</code> is the name of the interpreter that executes the Java program created by <code>javac</code> . The interpreter simulates a Java Virtual Machine on the host system. It interprets Java byte codes and executes them in the host system's <i>native code</i> .
appletviewer	<code>appletviewer</code> is a program to execute Java applets. You can also execute Java applets using a World Wide Web browser, such as Netscape.

You can retrieve these software tools from Sun Microsystem's web site. Point your browser at

`http://java.sun.com/j2se/`

These tools are part of the Java *Software Development Kit*, or *SDK*. As well as the programs noted above, the SDK includes a family of classes forming the Java *Application Programming Interfaces*, or *API*.

The version of Java used throughout these notes is 2. Since you are probably using these notes in a university or college course on Java, we'll assume that the programming environment either exists already or that an instructor is available to help.

## Simple Java Programs

OK, we're ready to begin. Figure 1 shows a very simple Java program called `Hello`, stored in a file called `Hello.java`. (The line numbers on the left are included for clarity. They are not actually in the file.)

```
1 public class Hello
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Hello, world");
6     }
7 }
```

Figure 1. Hello.java

At the moment, we aren't concerned with all the details of this program's operation. Here's what you must do to run this program:

1. Using a text editor, enter the program and save it in a disk file called `Hello.java`. Do not enter the line numbers shown in Figure 1. Line numbers appear in all demo programs in this text to assist in describing a program's operation. Java is case sensitive; so, be sure to enter the program exactly as shown.
2. Compile the program by entering

```
PROMPT><u>javac Hello.java
```

The characters underlined represent user input. The characters `PROMPT>` are how we will show the system prompt. This will vary from system to system. The program is compiled and a file called `Hello.class` is created. If you get any error messages from the compiler, use the editor to find and correct the error in the source program. Then re-compile.

3. Execute the program by entering

```
PROMPT><u>java Hello
```

This loads the Java interpreter, `java`, which in turn loads and executes the Java program stored in the file `Hello.class`. You should see the following output:

```
Hello, world
```

Congratulations! You have successfully entered, compiled, and executed your first Java program.

The seven-line listing in Figure 1 illustrates the basic framework for a Java program. The main ingredients are a *class* named `Hello` (line 1), a *function* or *method* named `main()` (line 3), and a method named `println()` (line 5). The `println()` method performs the critical task of outputting the message `Hello, world`. The message is outputted to `System.out`, which defaults to the host system's CRT display. `System.out` is also called the *standard output* or just *stdout*.

The program shows the *definition* of the `main()` method (lines 3-6) but only a *call* of the `println()` method (line 5). The definition of `println()` appears elsewhere — in Java libraries of classes and methods.

All Java applications must contain a class bearing the same name as the file read by the Java interpreter. Hence, there is a class named `Hello` in the file named `Hello.java`. Furthermore,

this class must have a method named `main()`. The `main()` method is the first method called when execution begins.

Two sets of braces appear in `Hello.java`. The first set (lines 2 and 7) encompasses the methods in the class `Hello`. In this example, there is only one method, `main()`. The second set (lines 4 and 6) encompasses the statements in the `main()` method. In this example, there is only one statement in the `main()` method.

The indentation in Figure 1 is completely cosmetic. We'll adhere to a style consistent with this throughout these notes; however, strictly speaking, this is not necessary. Computer programming is sufficiently abstract that you are well-advised to adopt a style similar to that in Figure 1. A good and consistent layout is extremely useful in understanding the overall strategy or flow in a program. In the extreme, Figure 2 illustrates an identical version of `Hello.java` that ignores indentation and runs the lines together. This program will compile and execute; but the source code is a mess to view.

```
public class Hello { public static void main(String[] args) {
System.out.println("Hello, world"); } }
```

Figure 2. `Hello.java` poorly formatted

Let's move on to an interesting variation of our first Java program. Figure 3 illustrates the source code for `Hello2.java`.

```
1 import java.io.*;
2
3 public class Hello2
4 {
5     public static void main(String[] args) throws IOException
6     {
7         // open keyboard for input (call it 'stdin')
8         BufferedReader stdin =
9             new BufferedReader(new InputStreamReader(System.in), 1);
10
11         // input a name from stdin (the keyboard)
12         System.out.print("Please enter your name: ");
13         String name = stdin.readLine();
14
15         // output a greeting to stdout (the CRT display)
16         System.out.println("Hello " + name);
17     }
18 }
```

Figure 3. `Hello2.java`

`Hello2.java` contains 18 lines of source code. You can create the program from scratch or copy it from the directory containing the demo programs for these notes. Compile the program by entering

```
PROMPT>javac Hello2.java
```

As you will see, `Hello2` is more interactive than our first demo program. A sample dialogue follows: (User input is underlined.)

```
PROMPT>java Hello2
```

```
Please enter your name: Jean
Hello Jean
```

There are many new ingredients in `Hello2` and a detailed discussion is beyond us at the moment. But, a quick walk-through won't hurt. As it turns out, Java requires a bit of effort to interact with a user through the keyboard.

In line 1, an `import` statement signals to the Java compiler that one or more methods appear in the program that must be obtained from the `java.io` library. These methods are the constructor method for `BufferedReader` class (line 8) and the constructor method for the `InputStreamReader` class (line 9). The result of calling the `BufferedReader()` constructor is to instantiate a new object of the `BufferedReader` class with the name `stdin`. Now, if the preceding three sentences are completely incomprehensible, don't worry. We are ahead of ourselves at this point. These concepts resurface later and we'll make a serious effort then to comprehend the incomprehensible. Right now, we just want to execute some programs to illustrate Java's capabilities. Let's put it this way: Lines 8 and 9 enable the program to obtain user input from the keyboard. The keyboard object is named `stdin`, for "standard input".

Lines 7, 11, and 15 begin with double slashes, `//`. These are comment lines. They serve only to provide explanations to the reader on the purpose of the program. When `//` appears anywhere on a line, the rest of the line is ignored by the compiler. An alternative method for comments is to use `/*` followed by `*/`. The compiler ignores anything in between. This method is useful for long comments that span several lines.

Line 12 outputs a prompt for the user to enter his or her name. Note that the `print` method in line 12 is `print()`, not `println()`. The suffix `"\n"` means that the printed message is followed by a new line. Without `"\n"`, the method prints the message without starting a new line, and user input occurs on the same line as the message.<sup>1</sup>

In line 13, the instance method `readLine()` appears. It acts on the `stdin` object — the keyboard. The effect is to get a line of input from the user and assign it to the `String` variable `name`.

In line 16, a message is printed on `System.out` — the CRT display. The message consists of two strings "concatenated", or joined, together. The two strings are `"Hello "`, which is hard-coded in the program, and the content of the string variable `name`, which was entered on the keyboard. The plus sign (+) is the concatenation operator.

Let's take the `Hello2` program one step further. Figure 4 shows the source code for `Hello3.java`.

---

<sup>1</sup> If the prompt `"Please enter your name: "` does not appear when the program begins execution, add a single line after line 12 containing `System.out.flush();` This forces characters printed using `print()` to appear immediately. Whether or not "flushing" is necessary is system-dependent. If it is necessary on your system, subsequent programs in this book using `print()` statements must be similarly modified. Note that flushing is never necessary with the more-common `println()` statement.

```

 1  /*****
 2  ** Hello3 - program to demonstrate simple loops
 3  **
 4  ** This program is the same as Hello2, except the message
 5  ** 'Java is fun!' is also output.  In fact, Java is so much fun,
 6  ** the message is output five times.
 7  **
 8  ** (c) Scott MacKenzie, 1999
 9  *****/
10  import java.io.*;
11
12  public class Hello3
13  {
14      public static void main(String[] args) throws IOException
15      {
16          // open keyboard for input (call it 'stdin')
17          BufferedReader stdin =
18              new BufferedReader(new InputStreamReader(System.in), 1);
19
20          // input a name from the keyboard
21          System.out.print("Please enter your name: ");
22          String name = stdin.readLine();
23
24          // output a greeting to the console display
25          System.out.println("Hello " + name);
26
27          // output 'Java is fun!' ten times
28          int i = 0;
29          while (i < 5)
30          {
31              System.out.println("Java is fun!");
32              i = i + 1;
33          }
34      }
35  }

```

Figure 4. Hello3.java

A comment block is shown preceding the program statements. The block begins with `/*` and ends with `*/`, and everything between is ignored by the compiler. In general, we will not show comment blocks in our demo programs reproduced here; however, they appear in the original files.

A sample dialogue with Hello3 follows:

```

PROMPT>java Hello3
Please enter your name: Bruce
Hello Bruce
Java is fun!
Java is fun!
Java is fun!
Java is fun!
Java is fun!

```

The main new ingredient is a loop to output the message `Java is fun!` five times (lines 19-24). Although we will not meet loops formally until later, they are an essential part of most Java programs. This example shows a "while" loop. An integer variable named `i` serves as a loop

counter. `i` is initialized to zero before the loop begins (line 19). The loop executes "while `i` is less than five" (line 20). The purpose of the loop is to repeatedly print the message `Java is fun!` The two statements within the loop are enclosed by braces (lines 21 and 24), with the effect that they are treated as a unit. One statement in the loop prints the message (line 22), while the other bumps-up the counter by one with each iteration (line 23).

The preceding is a brief walk-through of simple console-based Java programs. Input was text read from the keyboard, output was text printed on the CRT display. Next, we'll visit some of Java's more sophisticated features involving graphics, multi-media, and the internet.

## Java Applets (optional)

The preceding programs are examples of Java *applications*. A unique type of Java program is known as an *applet*. Applets are Java programs that can be executed by a web browser, such as Netscape's *Navigator* or Microsoft's *Internet Explorer*. This is easy to say, but, it's a tall order. The applet must exist on a server, but must execute remotely when retrieved by the browser. This is possible due to Java's unique architecture. Java "byte codes", mentioned earlier, are the primitive instructions of a "Java Virtual Machine". This machine is "virtual" because it does not exist. The purpose of the program, `java`, is to interpret Java byte codes and execute them on a physical machine. Thus, Java byte codes are an intermediate step between the high-level statements of the Java language and the low-level instruction set of a physical machine, such as an Intel *Pentium* microprocessor. The layer between the byte codes and the host system's native code is the Java Virtual Machine.

A Java-enabled browser has a built-in Java interpreter. It has the capability to interpret Java byte codes and execute them on a local machine. Well, almost. Because of the need to enforce security restrictions, not all features of the Java language are available for Java programs retrieved and executed from a web server. It would be unwise, for example, to permit a program retrieved from a remote web server to open and write disk files on a local machine. Think of the havoc this could cause. Applets represent a subset of the Java language designed specifically for transfer across the web for execution on a local machine. Some additional functionality for multi-media and other services is also supported for Applets.

With this introduction, let's see some Java applets in action. Figure 5 illustrates a simple applet that outputs the message "At your service, Master!" in a graphics window.

```

1  import java.awt.*;
2  import java.applet.*;
3
4  public class DemoApplet extends Applet
5  {
6      public void init()
7      {
8          setBackground(Color.lightGray);
9      }
10
11     public void paint(Graphics g)
12     {
13         Font f = new Font("Helvetica", Font.BOLD, 18);
14         g.setFont(f);
15
16         g.setColor(Color.blue);
17         g.drawString("At your service, Master!", 20, 50);
18
19         g.setColor(Color.black);
20         g.drawRect(0, 0, 249, 99);
21     }
22 }

```

Figure 5. DemoApplet.java

This program is compiled in the usual manner; however, since it is an applet, it is not executed using the program `java`. The applet must be referenced in a web document and executed using a Java-enabled browser, or using the `appletviewer` utility provided by Sun Microsystems. As a minimum, a file named `DemoApplet.html` containing the following two lines is necessary (see Figure 6)

```

<applet code = "DemoApplet.class" width = 250 height = 100>
</applet>

```

Figure 6. DemoApplet.html

With this, the applet is executed by opening `DemoApplet.html` with a Java-enabled web browser, or from the command line as follows:

```
PROMPT>>appletviewer DemoApplet.html
```

Figure 7 illustrates the result.



Figure 7. Output of DemoApplet applet

This program contains the definition of one class, `DemoApplet` in lines 4-22. Within `DemoApplet`, two methods are defined: `init()` in lines 6-9, and `paint()` in lines 11-21. Note that there is no method called `main()`. This underscores a key difference between applets and Java applications. Java applications are stand-alone programs. An applet is small program embedded inside another application, such as a web browser.

Within the `init()` method, the `setBackground()` method is called to set the background color of the applet to light gray (line 8). Within the `paint()` method, five methods are called. In line 13, a `Font` object named `f` is declared and instantiated. The instantiation occurs via the `Font` class constructor named `Font()`. The arguments within the parentheses specify the font family (Helvetica), style (bold), and size (18 point).

The next four methods all operate on an object named `g` — the graphics context of the `paint()` method. At this point, we needn't delve into the details. You can see in lines 14-20, however, that the font characteristics are set (line 14), the drawing color is set to blue (line 16), a message string is drawn at pixel coordinate  $x = 20$ ,  $y = 50$  (line 17), the drawing color is changed to black (line 19), and a rectangle is drawn at coordinate  $x = 0$ ,  $y = 0$  with *width* = 249 and *height* = 99 (line 20). Graphics coordinates are relative to the top-left corner of the graphics windows.

## Working With Graphics (optional)

Let's examine another applet that is a little fancier. The program `DemoGraphics` creates a bar chart showing the temperatures in four North American cities. The source code is shown in Figure 8.

```

1  import java.awt.*;
2  import java.applet.*;
3
4  public class DemoGraphics extends Applet
5  {
6      private static final String[] CITY =
7          { "Los Angeles", "Vancouver", "Toronto", "New York" };
8      private static final int[] TEMP = { 33, 28, 18, 22 };
9      private static final int XSIZE = 500;
10     private static final int YSIZE = 250;
11     private static final int GAP = 25;
12     private static final int RECT_WIDTH = XSIZE*2 / (TEMP.length * 3 + 2);
13     private static final int RECT_GAP = RECT_WIDTH / 2;
14     private static final double HEIGHT_FACTOR = (YSIZE - 2 * GAP) / 40.0;
15
16     public void paint(Graphics g)
17     {
18         g.setFont(new Font("Helvetica", Font.PLAIN, 12));
19
20         g.drawRect(0, 0, XSIZE - 1, YSIZE - 1);
21         g.drawLine(GAP, YSIZE - GAP, XSIZE - GAP, YSIZE - GAP);
22         g.drawLine(GAP, YSIZE - GAP, GAP, GAP);
23         for (int i = 0; i < CITY.length; i++)
24         {
25             int width = RECT_WIDTH;
26             int height = (int)(TEMP[i] * HEIGHT_FACTOR);
27             int x = GAP + RECT_GAP + i * (RECT_WIDTH + RECT_GAP);
28             int y = YSIZE - GAP - height;
29             g.setColor(Color.lightGray);
30             g.fillRect(x + 1, y + 1, width - 1, height - 1);
31             g.setColor(Color.black);
32             g.drawRect(x, y, width, height);
33             g.drawString(CITY[i], x, YSIZE - GAP + 15);
34             g.drawString(TEMP[i] + " deg C", x, YSIZE - GAP - height - 4);
35         }
36     }
37 }

```

Figure 8. DemoGraphics.java

This program is compiled using `javac` and executed using `appletviewer` or a browser. Remember, the `.class` file for applets is not directly executable. It must be referenced through a web document containing, as a minimum, the following two lines:

```

<applet code = "DemoGraphics.class" width = 500 height = 250>
</applet>

```

The output is shown in Figure 9.

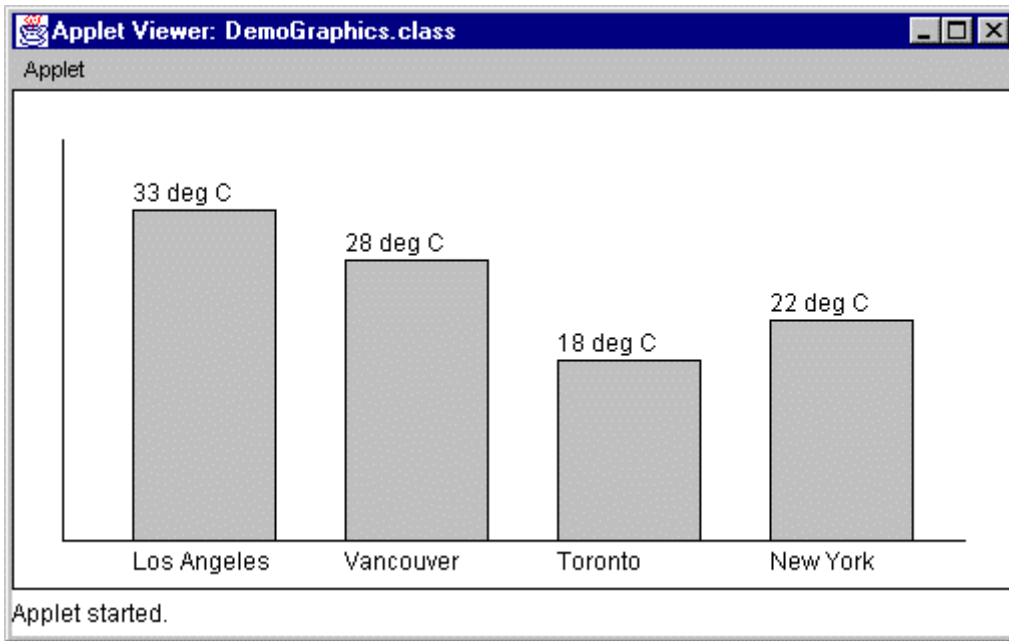


Figure 9. Output of DemoGraphics applet

As you can see in the 37 lines of source code in Figure 8, we have gone to quite a bit of trouble to position and label the four bars representing the temperature in each city. A lot of the effort was expended in "planning" the layout, as controlled through the defined constants in lines 6-14. Within the `for` loop in lines 23-35, these constants help in positioning the various graphical and text objects. The extra planning is well worth it. If the source file is edited and an extra city and temperature are added in lines 6 and 8, the output is nicely rearranged without any further changes.

### Working With Images (optional)

An important feature of Java applets is their ability to work with multi-media objects such as images or sounds. Figure 10 is an example using a scanned photograph as an image.

```

1  import java.awt.*;
2  import java.applet.*;
3
4  public class DemoImage extends Applet
5  {
6      private static final int STARTX = 25;
7      private static final int STARTY = 25;
8      private static final int WIDTH = 350;
9      private static final int HEIGHT = 230;
10
11     private static Image picture;
12
13     public void init()
14     {
15         picture = getImage(getDocumentBase(), "VariHall.jpg");
16     }
17
18     public void paint(Graphics g)
19     {
20         g.drawImage(picture, STARTX, STARTY, WIDTH, HEIGHT, this);
21         g.drawString("Vari Hall", STARTX, STARTY + HEIGHT + 12);
22     }
23 }

```

Figure 10. DemoImage.java

This program is compiled using `javac` and executed using `appletviewer` or a browser. Remember that the `.class` file for applets is not directly executable. It must be referenced through a web document containing, as a minimum, the following two lines:

```

<applet code = "DemoImage.class" width = 400 height = 280>
</applet>

```

The output is shown in Figure 11.

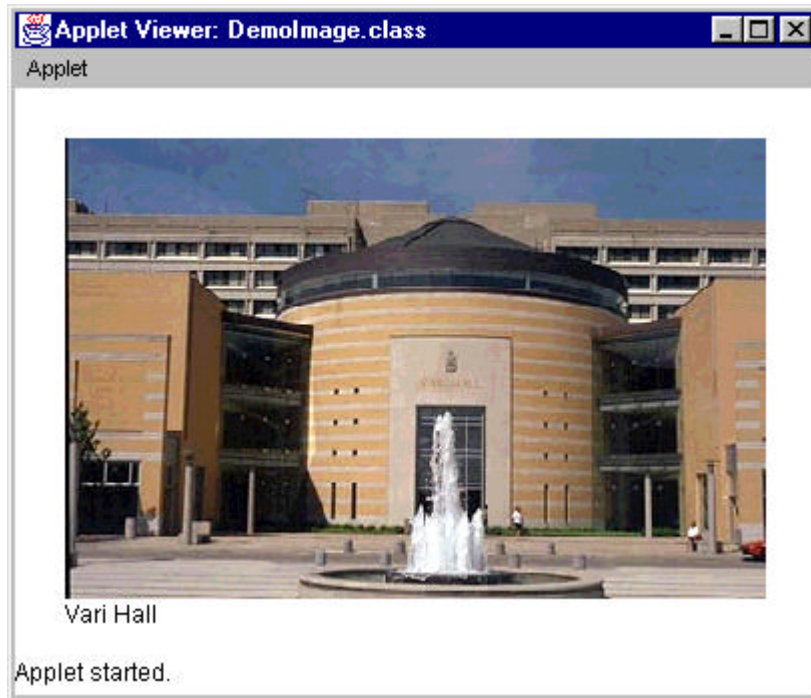


Figure 11. Output of DemoImage applet

The structure of the DemoImage applet is similar to that of DemoApplet. In line 11, an Image object named picture is declared. In line 15, the image is retrieved using the getImage() method. getImage() requires two arguments: the getDocumentBase() method retrieves the directory path of the applet on the server, and the string "VariHall.jpg" identifies the file containing the image. The image is drawn in the applet window in line 20 using the drawImage() method. Lines 6-9 contain four defined constants that assist in positioning objects in the graphics window. The WIDTH and HEIGHT constants specify the dimensions of the image.

### Working With Sound (optional)

Figure 12 is an example of an applet that plays a sound.

```

1  import java.awt.*;
2  import java.awt.event.*;
3  import java.applet.*;
4
5  public class DemoSound extends Applet implements ActionListener
6  {
7      private static AudioClip tone;
8      private static Button beep;
9
10     public void init()
11     {
12         tone = getAudioClip(getDocumentBase(), "sounds\\9.au");
13         beep = new Button("Beep");
14         beep.addActionListener(this);
15         add(beep);
16     }
17
18     public void actionPerformed(ActionEvent ae)
19     {
20         if (ae.getSource() == beep)
21             tone.play();
22     }
23 }

```

Figure 12. DemoSound.java

Instead of simply playing the sound once, a button is used to play the sound at the request of the user. This as illustrated in Figure 13.



Figure 13. Output of DemoSound applet

Each time the Beep button is selected with a mouse click, the sound is heard. Let's have a quick look at the source code. In line 7, an `AudioClip` object named `tone` is declared. The file containing the sound is retrieved in line 12 using the `getAudioClip()` method. A `Button` object named `beep` is declared in line 8. The button is instantiated in line 14 and assigned the label "Beep".

Since the button is activated by a mouse click, some extra work is necessary to link mouse clicks to the activation of the sound object. In fact, the `DemoSound` applet is an "event-driven" program. The program is setup to enable and respond to events, rather than to execute statements in sequential order. Whenever a mouse click occurs, the `actionPerformed()` method executes (lines 18-22). If the click occurred on the button (line 20), the `play()` method is invoked to activate the sound object, `tone` (line 21).

## Graphical User Interfaces (optional)

Our final demo program in this section is an application program, rather than an applet. It uses a *graphical user interface*, or GUI, and makes extensive use of Java's *advanced windowing toolkit*,

known as the "awt". The program implements a very simple drawing package, capable of drawing lines, rectangles, and ovals. The complete listing is shown in Figure 14.

```

1  import java.awt.*;
2  import java.awt.event.*;
3
4  class DemoGuiDraw
5  {
6      public static void main(String[] args)
7      {
8          DrawWindow dw = new DrawWindow("Demo Gui Draw");
9          dw.setSize(DrawWindow.SIZE, DrawWindow.SIZE);
10         dw.setVisible(true);
11     }
12 }
13
14 class DrawWindow extends Frame implements
15     WindowListener,
16     MouseListener,
17     MouseMotionListener,
18     ActionListener
19 {
20     static final int SIZE = 300;
21
22     int mode = -1;        // default mode (do nothing)
23     int newMode = -1;
24
25     String[] modeLabel = { "Rectangle", "Oval", "Line" };
26     Button[] modeButton = new Button[modeLabel.length];
27
28     int x1, y1, x2, y2, tempx, tempy;
29
30     public DrawWindow(String s)
31     {
32         super(s);
33         setBackground(Color.white);
34         setLayout(null);
35         initializeButtons();
36         addMouseListener(this);
37         addMouseMotionListener(this);
38         addWindowListener(this);
39     }
40
41     private static final int STARTX = 30;
42     private static final int STARTY = 30;
43     private static final int WIDTH = 70;
44     private static final int HEIGHT = 20;
45     private static final int HGAP = 10;
46     private static final int VGAP = 7;
47
48     public void initializeButtons()
49     {
50         int x = STARTX;
51         int y = STARTY;
52         for (int i = 0; i < modeLabel.length; i++)
53         {
54             modeButton[i] = new Button(modeLabel[i]);
55             modeButton[i].setLocation(x, y);
56             modeButton[i].setSize(WIDTH, HEIGHT);
57             modeButton[i].setBackground(Color.lightGray);
58             add(modeButton[i]);
59             modeButton[i].addActionListener(this);
60             x += WIDTH + HGAP;
61             if (x > SIZE - (WIDTH + HGAP)) // start a new row
62                 {

```

```

63         x = STARTX;
64         y += HEIGHT + VGAP;
65     }
66 }
67 }
68
69 public void actionPerformed(ActionEvent event)
70 {
71     Object source = event.getActionCommand();
72     for (int i = 0; i < modeLabel.length; i++)
73         if (source.equals(modeLabel[i]))
74             newMode = i;
75     if (mode >= 0 && mode < modeLabel.length)
76         modeButton[mode].setBackground(Color.lightGray);
77     modeButton[newMode].setBackground(Color.cyan);
78     mode = newMode;
79 }
80
81 public void mouseClicked(MouseEvent me) { }
82 public void mouseEntered(MouseEvent me) { }
83 public void mouseExited(MouseEvent me) { }
84 public void mouseMoved(MouseEvent me) { }
85
86 // ----- M O U S E   P R E S S E D -----
87 public void mousePressed(MouseEvent me)
88 {
89     x1 = me.getX();
90     y1 = me.getY();
91     tempx = x1;
92     tempy = y1;
93 }
94
95 // ----- M O U S E   D R A G G E D -----
96 public void mouseDragged(MouseEvent me)
97 {
98     Graphics g = getGraphics();
99     g.setColor(Color.black);
100    g.setXORMode(Color.white);
101    x2 = me.getX();
102    y2 = me.getY();
103
104    // draw rectangle mode
105    if (mode == 0)
106    {
107        // draw over old rectangle to erase it (XOR mode)
108        g.drawRect(Math.min(x1, tempx), Math.min(y1, tempy),
109                  Math.abs(x1 - tempx), Math.abs(y1 - tempy));
110
111        // draw new rectangle
112        g.drawRect(Math.min(x1, x2), Math.min(y1, y2),
113                  Math.abs(x1 - x2), Math.abs(y1 - y2));
114    }
115
116    // draw oval mode
117    else if (mode == 1)
118    {
119        g.drawOval(Math.min(x1, tempx), Math.min(y1, tempy),
120                  Math.abs(x1 - tempx), Math.abs(y1 - tempy));
121        g.drawOval(Math.min(x1, x2), Math.min(y1, y2),
122                  Math.abs(x1 - x2), Math.abs(y1 - y2));
123    }
124
125    // draw line mode

```

```

126     else if (mode == 2)
127     {
128         g.drawLine(x1, y1, tempx, tempy);
129         g.drawLine(x1, y1, x2, y2);
130     }
131
132     tempx = x2;
133     tempy = y2;
134 }
135
136 // ----- M O U S E   R E L E A S E D -----
137 public void mouseReleased(MouseEvent me)
138 {
139     Graphics g = getGraphics();
140     x2 = me.getX();
141     y2 = me.getY();
142
143     // draw rectangle
144     if (mode == 0)
145         g.drawRect(Math.min(x1, x2), Math.min(y1, y2),
146                   Math.abs(x1 - x2), Math.abs(y1 - y2));
147
148     // draw oval
149     else if (mode == 1)
150         g.drawOval(Math.min(x1, x2), Math.min(y1, y2),
151                  Math.abs(x1 - x2), Math.abs(y1 - y2));
152
153     // draw line
154     else if (mode == 2) // line
155         g.drawLine(x1, y1, x2, y2);
156
157     return;
158 }
159
160 public void windowClosed(WindowEvent we)      { }
161 public void windowDeiconified(WindowEvent we) { }
162 public void windowIconified(WindowEvent we)  { }
163 public void windowActivated(WindowEvent we)  { }
164 public void windowDeactivated(WindowEvent we) { }
165 public void windowOpened(WindowEvent we)     { }
166
167 public void windowClosing(WindowEvent we)
168     { System.exit(0); }
169 }

```

Figure 14. DemoGuiDraw.java

An example of DemoGuiDraw running is shown in Figure 15. A simple drawing illustrates the program's capabilities.

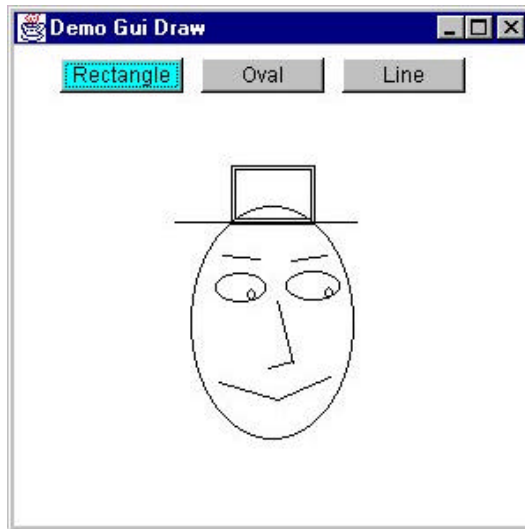


Figure 15. DemoGuiDraw window containing a drawing

The 169 lines of source code in `DemoGuiDraw` cover a lot of territory. What we are after here is "the big picture". The details will unfold during the rest of this text.

Here's a brief walk-through of the structure of `DemoGuiDraw`. The program contains the definition of two classes: `DemoGuiDraw` (line 4-12) and `DrawWindow` (lines 14-169). `DemoGuiDraw` contains the `main()` method that executes when the program begins (lines 6-11). Within the `main()` method, only one object is instantiated, a `DrawWindow` object named `dw` (line 8). The next two lines set the size of `dw` and set its visibility attribute to `true`.

Most of the work occurs in the methods in the `DrawWindow` class. First and foremost is the constructor method, `DrawWindow()`, defined in lines 30-39. The `DrawWindow()` constructor method is called in line 8 within the `main()` method. This method calls other methods to initialize buttons, initialize event listeners, and to exit the application. Like `DemoSound` presented earlier, `DemoGuiDraw` is an example of an event-driven program. The event listeners are routines that execute only in response to events, such as moving the mouse, or clicking a mouse button.

For the moment, you are encouraged simply to copy, compile, and execute `DemoGuiDraw`. And, of course, play with it by showing off your drawing talents.

## Concepts

This concludes our brief getting-started tour of Java. Although we've avoided detailed discussions, you probably noticed several concepts that keep popping up. In Java, we talk a lot about "classes", "objects", and "methods". Some of the methods we used, we also defined, such as `actionPerformed()` in `DemoSound.java` (see Figure 12). On the other hand, some of the methods we used were defined elsewhere, such as `println()` in `Hello.java` (see Figure 1). The methods defined elsewhere are part of Java's API in the Java development kit (jdk), provided by Sun Microsystems.

All Java programs contain the definition of at least one class, and within this class at least one method is defined — the method that first executes when the program begins. The Java language is a large hierarchy of classes. An object is an instance of a class. The process of creating an object, we refer to as "instantiating an object". A good example is the button in

DemoSound.java (see Figure 12). There is a class called Button. We instantiated an object of the Button class and named it beep.

Objects are sophisticated entities. In fact, objects are composed of primitive data types, such as integers, real numbers, and characters and the method to operate on these. In the next section, we will examine Java's primitive data types.

### **Summary**

These notes introduced the following key concepts:

- Java is an object-oriented programming language.
- A set of tools to develop Java programs is available free from Sun Microsystem's web site. The tools are part of the SDK - Software Development Kit.
- The tools used in this book are `javac` (the Java compiler), `java` (the Java interpreter), and `appletviewer` (the viewer for Java applets).
- A Java program is created using an editor and saved in a file with ".java" appended to the file name. The file contains the source code for a Java program.
- A Java program is compiled using `javac` - the Java compiler. The result is a ".class" file containing Java byte codes.
- A Java application program is executed using `java` - the Java interpreter.
- The Java interpreter simulates a Java Virtual Machine. It interprets Java byte codes and executes them using the native code instructions of the host machine.
- A Java applet referenced in an html file may be viewed using `appletviewer` - the Java applet viewer.
- Java applets may display graphics, images, and they can play sounds.
- Java is a full-fledged programming language capable of implementing Graphical User Interfaces.

