Java Primer

Preface

The enclosed notes serve as a ground-level introduction to programming in the language Java. Six chapters exist at present, but others are contemplated. Unfortunately, there is frequent mention of Appendix C, but it does not exist at present. The information planned for Appendix C will be a subset of the JAVA API document, which is available at

`http://java.sun.com/products/jdk/1.2/docs/api/index.html`

Good luck, S.M. 27-Jan-00

Table of Contents

Chapter 1

Getting Started

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 *DOS* window, you can use `EDIT` or `Notepad`. If you're working in a unix environment (e.g., *linus*, *Solaris*), you can use `vi` or `pico`. However, we'll leave the choice of editor to you. The file you create with an editor contains the *source code* for a Java program. The file must have "`.java`" appended to the filename.
`javac`	`javac` is the name of the Java compiler. It is the program that translates a Java source code program into Java *byte codes*. The file created by `javac` has "`.class`" as the filename suffix. The information in the "`.class`" file is often called *object code* to distinguish it from the source code in the "`.java`" file.
`java`	`java` is the name of the interpreter that executes the Java program created by `javac`. The interpreter simulates a Java Virtual Machine on the host system. It interprets Java byte codes and executes them in the host system's *native code*.
`appletviewer`	`appletviewer` 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 S*oftware 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>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>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 an 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 "ln" means that the printed message is followed by a new line. Without "ln", the method prints the message without starting a new line, and user input occurs on the same line as the message.[1]

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`.

---

[1] 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.

```
    /**************************************************************
    ** Hello3 - program to demonstrate simple loops
    **
    ** This program is the same as Hello2, except the message
    ** 'Java is fun!' is also output.  In fact, Java is so much fun,
    ** the message is output five times.
    **
    ** (c) Scott MacKenzie, 1999
    **************************************************************/
 1  import java.io.*;
 2
 3  public class Hello3
 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 the keyboard
12        System.out.print("Please enter your name: ");
13        String name = stdin.readLine();
14
15        // output a greeting to the console display
16        System.out.println("Hello " + name);
17
18        // output 'Java is fun!' ten times
19        int i = 0;
20        while (i < 5)
21        {
22           System.out.println("Java is fun!");
23           i = i + 1;
24        }
25     }
26  }
```

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 mininum, 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.

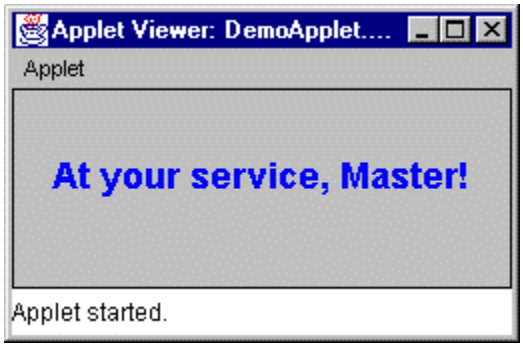


Figure 7. Output of `DemoApplet` applet

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.

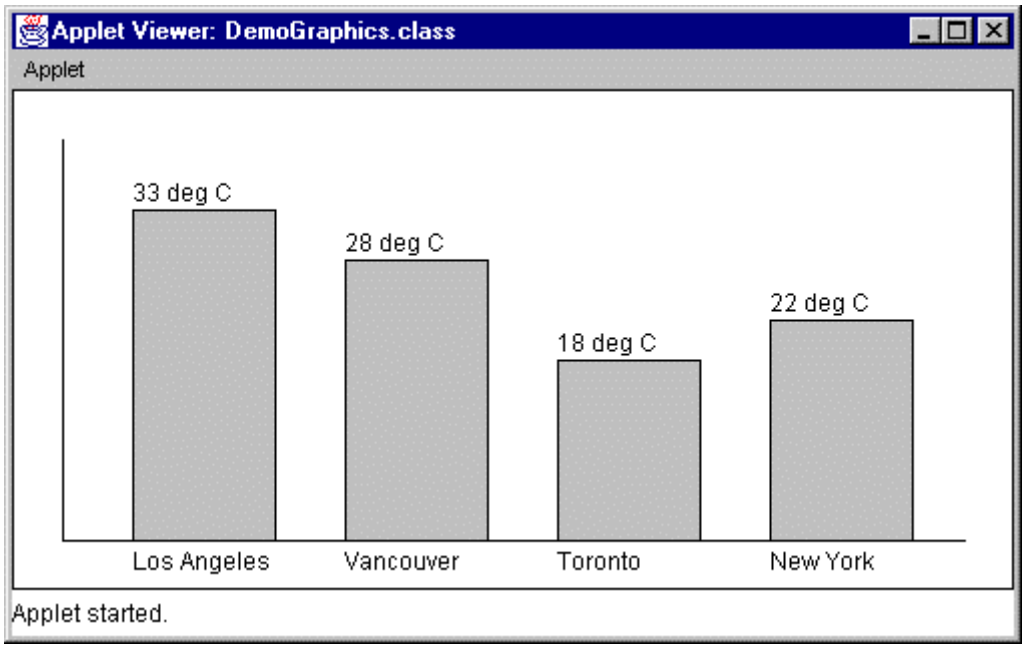


Figure 9. Output of `DemoGraphics` applet

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.

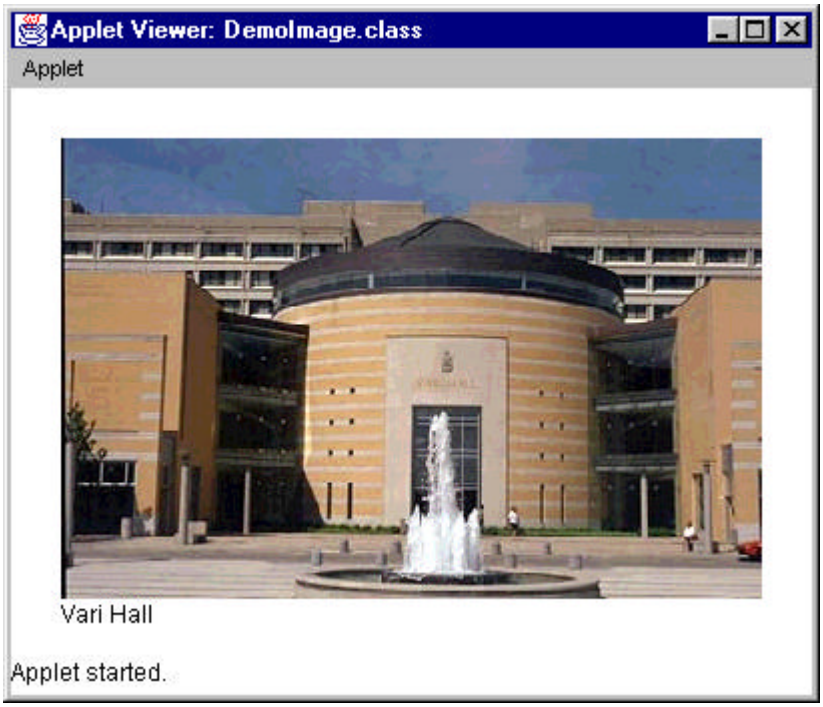


Figure 11. Output of `DemoImage` applet

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 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 rectange 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.

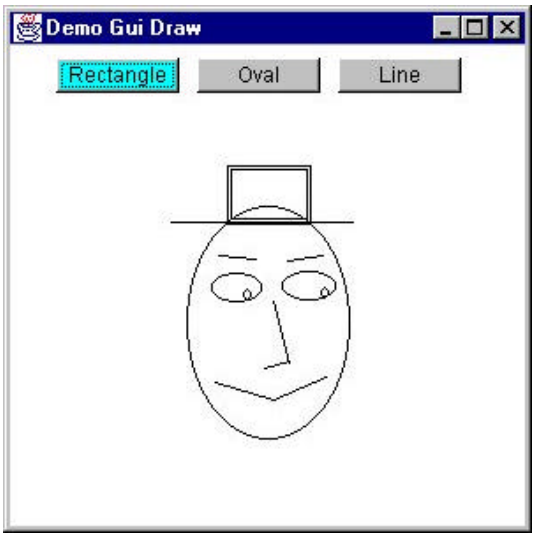


Figure 15. `DemoGuiDraw` window containing a drawing

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.

Chapter 2

Program Elements

Primitive Data Types

The building blocks for all objects in Java are the *primitive data types* in the Java language. In this section, we'll meet Java's primitive data types and learn how to name, store, retrieve, assign, and operate on these types. We'll introduce the concept of a *variable*. A variable provides a convenient way to name an instance of a data type. For a variable to exist, however, it first must be *declared*. For a variable to be used, it first must have a value *assigned* to it. To perform meaningful programming tasks with variables, we need a set of *operations* that can be performed with variables. Strings are not primitive data types in Java; however, their treatment is so special that we include them in this chapter. These and other concepts are introduced in this chapter.

Java includes the following eight primitive data types:

`int`	an integer number (32 bits)
`long`	an integer number (64 bits)
`short`	an integer number (16 bits)
`byte`	an integer number (8 bits)
`double`	a real number (64 bits)
`float`	a real number (32 bits)
`char`	a character (16 bits)
`boolean`	a boolean (1 bit)

The first six are examples of numbers, and we'll begin with these.

Integers and Floating-Point Numbers

There are two general categories of numbers: integers and real numbers. Integers are whole numbers and have no fractional or decimal component. So, 2 is an integer, but 2.718 is not. Many quantities are adequately represented by integers. For example, the number of passengers in a car could be 4 or 5, but not 4.5. Many quantities, however, need a fractional or decimal component, such as the ratio of the circumference of a circle to its diameter, known as *pi*. For these quantities, a real number is required. Because of the way numbers are stored and represented in computer systems, real numbers are more commonly called *floating-point numbers*.

In the preceding list, the first four data types are integers and the next two are floating-point numbers. The most common type of integer is `int`, but Java also supports long integers (`long`), short integers (`short`), and byte integers (`byte`). The most common type of floating-point number is `double`, for "double-precision", but Java also supports single-precision floating-point numbers (`float`).

As you might guess, the variations on integers and floating-point numbers differ in their size and internal representation on a computer. From a numeric perspective, they differ in their *range* and *precision*. Range is the spread between the smallest and largest quantity represented. Precision is the closeness to a single quantity that can be represented. For integers, the precision is always one. For floating-point numbers, the precision is the number of significant, or meaningful, digits

that can be represented. To use *pi* as an example, the three digits 3.14 may be adequate to some, but mathematicians can quantify *pi* with hundreds of digits of precision. Unfortunately, such precision is not possible for the primitive type `double` in Java. With a `double`, we get about 15 digits of precision. So, the best we can do for *pi* is about 3.141592653589793.[1] *Table 1* summarizes the range and precision for Java's integer and floating-point data types.[2]

Table 1. Range and Precision for Primitive Data Types

		Range		
Type	Bits	Smallest	Largest	Precision
`int`	32	-2147483648	2147483647	1
`double`	64	4.9E-324	1.7976931348623157E308	15 digits
`long`	64	-9223372036854775808	9223372036854775807	1
`short`	16	-32768	32767	1
`byte`	8	-128	127	1
`float`	32	1.4E-45	3.4028235E38	7 digits

The ranges for floating-point numbers are shown in scientific notation in *Table 1*, where "E*n*" means 10^n. A `double` variable can be as small as -4.9×10^{324} or as large as $+1.8 \times 10^{308}$ with about 15 digits of precision.

The number of bits in each primitive data type determines the memory required to hold data of that type. To store one thousand numbers, each as a `double`, for example, $1{,}000 \times 64 =$ 64,000 bits = 8,000 bytes of memory are needed. If only a few digits of precision are necessary, storing each value as a `float` is a space-saving option to consider. In most cases today, memory is plentiful and cheap; so, conserving space may not be an issue.

In comparing integers with floating-point numbers, the speed of operations should be considered. Primitive operations such as multiplication and division are considerably faster with integers than with floating-point numbers, and this should be considered when choosing a data type to store information. Today's desktop computers have extremely high frequency CPUs, so execution speed may not be an issue. Bear in mind, however, that memory requirements are important for very large quantities of data, and execution speed is important for programs that perform vast and complex computations.

Characters

Besides numbers, computers must store and manipulate characters, such as letters of the alphabet, digits, punctuation symbols, etc. The primitive data type `char` serves this purpose. In Java a character is coded as the symbol for the character enclosed in single quotes. Examples include `'A'`, `'a'`, `'9'`, `'%'`, and `'='`.

Since memory locations store a series of bits, each with state 0 or state 1, there is no obvious way to store characters in a computer's memory. The trick is to devise a coding system whereby each

---

[1] This is the value assigned to the constant `Math.PI` is Java's `Math` class in the `java.lang` package.

[2] The values in *Table 1* were obtained by printing the defined constants in Java's wrapper classes. See the program `FindRanges.java` for more details.

character is represented by a certain pattern of bits. A well-established coding system for characters is the *American Standard Code for Information Interchange*, or *ASCII* (pronounced *ass-key*). ASCII is a 7-bit coding system. Of its $2^7 = 128$ codes, 95 are for graphic symbols (e.g., letters, digits, punctuation), and 33 are for control characters (e.g., enter, tab, delete, end-of-file).

Unfortunately, 7-bit ASCII codes provide little room for expansion. International requirements of computing technology necessitate a coding system that can accommodate all the major scripts of the world. A 16-bit coding system known as the *Unicode* Standard was developed by an industry consortium to meet such needs. The ASCII character set is a subset of Unicodes, occupying the first 128 positions in the Unicode set. For example, the character `'A'` is represented by the Unicode `\u0041`. The code `0041` is the hexadecimal value corresponding to the letter `A`. This is equivalent to the decimal value 65.

Escape Sequences

Besides graphic characters that are printable, the `char` data type can hold values that control input/output devices or modify the appearance or layout of information. These are known as *control characters*. Since control characters are not printable, they cannot be scripted as a symbol enclosed in single quotes, like other characters. Instead they are represented as *escape sequences*. An escape sequence is a two-character pattern beginning with the backslash character (\). The most common Java escape sequences are given in Table 2.

Table 2. Java Escape Sequences

Escape Sequence	Interpretation
`\b`	backspace
`\t`	tab
`\n`	newline
`\r`	carriage return
`\"`	"
`\\`	\
`\'`	'

So, the tab character is coded as `'\t'`, the newline character as `'\n'`, and so on.

Booleans

A `boolean` data type can take on one of two values: `true` or `false`. Since only two states must be represented, only 1 bit of memory storage is required to store a boolean value. The term "boolean" is used in deference to the 19th century mathematician George Boole who pioneered the branch of mathematics known as *Boolean algebra* or *combinatorics*. Note that the terms `true` and `false` are reserved words in Java and cannot be used for other purposes.

Variables and Constants

We are now ready to put Java's primitive data types to work. A primitive data type may appear in a program two ways: as a *variable* or as a *constant*. A constant, sometimes called a *literal*, is simply an explicit instance of a value. For example, `3` is an integer constant, `3.14` is a floating point constant, `'a'` is a character constant, and `true` is a boolean constant. Integer constants

can be expressed in hexadecimal notation, if desired. For example `0x00FF` is an integer constant equal to `255`. The pattern "`0x`" means that what follows is interpreted in hexadecimal notation.

A variable is an abstraction that represents a value. A key difference between a constant and a variable is that a constant, once coded in a program, cannot change. A variable holds a value, but this value can change as a program executes.

Before a variable is used, it must be *declared* and it must be *initialized* to a value. The following four Java statements declare integer, floating-point, character, and boolean variables:

```
int count;
double temperature;
char c;
boolean status;
```

Each line above is an example of a Java *statement*. Note that a statement ends with a semicolon. In fact, each line above is a particular type of Java statement known as a *declaration*. The first line declares a variable named `count` of type `int`. The second line declares a variable named `temperature` of type `double`. And so on. The variable's name is chosen by the programmer. Obviously, one should choose names that suggest the variable's purpose. The name "temperature" gives a pretty good indication of what the variable represents.

So, once declared, how do we initialize a variable to a value? The following eight program statements show the declaration of variables followed immediately by the initialization of the variables:

```
// declare four variables
int count;
double temperature;
char c;
boolean status;

// initialize the four variables to values
count = 7;
temperature = 98.6;
c = 'A';
status = true;
```

Note that a single character is coded as the character symbol enclosed in single quotes. Declaration and initialization can be combined:

```
int count = 7;
double temperature = 98.6;
char c = 'A';
boolean status = true;
```

It is also possible to declare and initialize more than one variable at the same time, for example

```
int i = 3, j = 4, k = 5;
```

A declaration can appear anywhere in a Java program, provided the declaration precedes the first use of the variable in an expression. There are different opinions on where declarations should appear. Some feel they should be grouped together at the beginning of a method or a block of code. Others feel they should appear close to their first use. It is largely a matter of personal style, and we'll not adhere strictly to either approach in these notes.

Comments

Comments are notations included in a source program to help explain an operation. There are two ways to include comments in Java programs. When two forward slash characters (`//`) appear on a line, the rest of the line is a comment and is ignored by the compiler. This technique is useful for short comments, such as a one-line comment or a comment appended to the end of a program statement. Long comments, perhaps spanning many lines of source code, can use "`/*`" and "`*/`" to delimit the comment, such as

```
/* this is a comment */
```

Comments that span many lines are usually called a *comment block*.

Identifiers and Reserved Words

Whenever a programmer creates a name for a variable or other component of a program (e.g., the name of a class), the result is an *identifier*. So, in the preceding examples, `temperature` was an identifier, `status` was an identifier, and so on. Clearly, there must be rules on what constitutes a legal or illegal identifier. And there are. A Java identifier can consist of any letters or digits, as well as an underscore (_), or a dollar sign ($). A digit character, however, cannot be the first character in an identifier. The following are examples of legal identifiers:

```
number_of_items
inputMode
X99
temp$filename$suffix
```

The following are examples of illegal identifiers:

`99gretzky`	(Illegal: can't start with a digit)
`odd-ball`	(Illegal: can't use a dash)
`this.that`	(Illegal: can't use a period)

There is no size restriction on identifiers, so if you want an identifier with a hundred characters, you can have it. Java is case sensitive, however, so beware that the following two identifiers are different:

```
pressure
Pressure
```

As a programmer, you can make-up identifiers that are as cryptic or as meaningful as you wish. Obviously, well-chosen identifiers help to clarify program statements. Besides the few rules noted above, an identifier must not conflict with Java's *reserved words*. Think of the problems if an integer variable were named `float`. The statements

```
int float;  // wrong!
float = 3;  // wrong!
```

would make the compiler's job quite difficult. Java's 59 reserved words are summarized in *Figure 1*.

abstract	boolean	break	byte	byvalue
case	cast	catch	char	class
const	continue	default	do	double
else	extends	false	final	finally
float	for	future	generic	goto
if	implements	import	inner	instanceof
int	interface	long	native	new
null	operator	outer	package	private
protected	public	rest	return	short
static	super	switch	synchronized	this
throw	throws	transient	true	try
var	void	volatile	while	

Figure 1. Java's reserved words

Naming Conventions

Identifiers are used for more than just the names of variables. They are also used for the names of methods, classes, and defined data constants. For each of these, naming conventions exist, and adhering to these will contribute to making your code understandable. Java's naming conventions are listed in *Figure 2*.

Type of Identifier	Examples	Convention
variable	temperature moreData	• starts with lowercase character • multiple words joined together • first letter of words (except first) in uppercase
method	length() toLowerCase()	• same rules as for variables • parentheses distinguish methods from variables • exception: constructor methods begin with an uppercase character
class	String MenuBar	• starts with uppercase character • multiple words joined together • first letter of words in uppercase
constant	FACTOR SCREEN_WIDTH	• all characters in uppercase • multiple words joined with underscore character

Figure 2. Conventions for naming identifiers

Operators

Representing and storing primitive data types is, of course, essential for any computer language. But, so, too, is the ability to perform operations on data. Java supports a comprehensive set of *operators* to facilitate adding, subtracting, multiplying, dividing, comparing, etc. Let's start with a simple demonstration program. Figure 1 contains the listing for a program that adds one and one to get two.

```
 1  public class Numbers
 2  {
 3     public static void main(String[] args)
 4     {
 5        int x;
 6        x = 1;
 7        int y = 1;
 8        int z = x + y;
 9        System.out.print("1 + 1 = ");
10        System.out.println(z);
11     }
12  }
```

Figure 1. `Numbers.java`

The overall structure of this program is identical to the Java applications we saw in the previous chapter. Now, however, we are operating on data. When this program executes, it generates the following output:

```
1 + 1 = 2
```

This is pretty trivial, but don't be fooled. There are many aspects of the Java language sitting before us in Figure 1. Let's walk through the program's use of data types and operators.

In line 5, an `int` variable named `x` is declared. The effect is to set aside storage for `x`, but the storage is not initialized with a value (see Figure 2a). In line 6, the variable `x` is assigned the value 1. The equals sign (=) is known as the *assignment* operator. The effect is to place the value 1 into the storage location set aside for the variable `x` (see Figure 2b).

Figure 2. (a) A variable `x` is declared, but not initialized
(b) a variable `x` is initialized with the value 1.

In line 7, we see declaration combined with assignment. A new `int` variable, `y`, is declared and initialized with the value 1.

Line 8 contains another example of declaration combined with assignment. An `int` variable `z` is declared and assigned the value `x + y`. The plus sign (+) is the *addition* operator. Since line 8 contains two operators, there is a need to process the operations in a certain order. This process is governed by Java's *precedence of operators*. As you might guess, addition takes precedence over assignment; so, the expression `x + y` is evaluated first and then the result is assigned to `z`. The result is printed in lines 9 and 10.

Java also includes operators for *subtraction* (`-`), *multiplication* (`*`), and *division* (`/`). Where these and other operators are mixed, precedence must be carefully considered. Let's examine precedence of operators in more detail. The program `Operators.java` (Figure 3) shows subtraction and multiplication combined in a single expression.

```
 1  public class Operators
 2  {
 3     public static void main(String[] args)
 4     {
 5        int i =  12 - 5  * 3;
 6        int j = (12 - 5) * 3;
 7
 8        System.out.println(i);
 9        System.out.println(j);
10     }
11  }
```

Figure 3. `Operators.java`

The output of this program is

```
-3
21
```

In line 5, an `int` variable `i` is declared and assigned the result of the expression `12 - 5 * 3`. If we evaluate the expression left-to-right the answer is 21, which is wrong because multiplication takes precedence over subtraction. The expression `5 * 3` is evaluated first and the result, 15, is then subtracted from 12 to yield the correct answer of -3. To perform the subtraction first, if desired, parentheses are added as shown in line 6. Parentheses override the natural precedence of operators.

Division and multiplication are of equal precedence, as are addition and subtraction. When arithmetic operators of the same precedence appear together, they are evaluated left to right. So, the expression `2 + 3 - 5 - 6` equals -6 and the expression `30 * 4 / 3 / 8` equals 5.

The Remainder Operator

When dividing integers, the decimal portion of the result, or the remainder, if any, is lost. So, the expression `13 / 5` equals 2. The remainder after division of integers can be obtained using the *remainder* (`%`) operator. (The remainder operator is sometime called the *modulus* or *mod* operator). So, the expression `13 % 5` equals 3, because 13 divided by 5 is 2 with a remainder of 3. The program `Days` (Figure 4) demonstrates the mod operator in action.

```
 1  public class Days
 2  {
 3     public static void main(String[] args)
 4     {
 5         int seconds =  1000000; //one million
 6
 7         int days      = seconds / (24 * 60 * 60);
 8         int remainder = seconds % (24 * 60 * 60);
 9         int hours     = remainder / (60 * 60);
10         remainder     = remainder % (60 * 60);
11         int minutes   = remainder /  60;
12         remainder     = remainder %  60;
13
14         System.out.println(seconds   + " seconds = ");
15         System.out.println(days      + " days");
16         System.out.println(hours     + " hours");
17         System.out.println(minutes   + " minutes, and");
18         System.out.println(remainder + " seconds");
19     }
20  }
```

Figure 4. `Days.java`

The `Days.java` program computes the number of days, hours, minutes, and seconds in one million seconds. The output is

```
1000000 seconds =
11 days
13 hours
46 minutes, and
40 seconds
```

In line 7, the number of days in one million seconds is computed by dividing the `int` variable `seconds` by (24 * 60 * 60). The result is 11. The arithmetic is for integers, so the remainder is lost. The remainder is retrieved in line 8 by performing the same operation again, except using the mod operator. The result — the remainder — is assigned to the `int` variable `remainder`. The number of hours is computed by dividing `remainder` by (60 * 60) in line 9. A similar process is repeated for hours, minutes, and seconds (lines 10-12).

When one or both values are negative, extra caution is warranted. The operations must obey the following rule:

```
(x / y) * y + x % y == x
```

So, for example

```
 7 %  2      equals 1
 7 % -2      equals 1
-7 %  2      equals -1, and
-7 % -2      equals -1.
```

Although much less used, the remainder operator also works with floating-point numbers. In this case, it returns the remainder after "even" division, for example, the expression `8.0 % 2.5` returns 0.5.

The % operator has the same precedence as multiplication and division.

Precedence of Operators

Although Java includes many more operators, let's summarize what we have learned thus far about the precedence of operators (see *Table 1).*

Table 1. Precedence of Operators

Precedence	Operator(s)	Operation	Association
highest	`( )`	override natural precedence	Inner to outer
next highest	`* / %`	multiplication, division, remainder	L to R
next highest	`+ -`	addition, subtraction	L to R
lowest	`=`	assignment	R to L

The association is left-to-right for most operators. One exception is the assignment operator. In the event of more than one assignment operator in a single statement, the association is right to left. Consider, for example, the following statements:

```
int x = 2;
int y = 3;
int z;
int a = z = x + y;
```

In final line, the expression `x + y` is evaluated first, then the result is assigned to `z`. Finally, the value of `z` is assigned to `a`.

Promotion of int to double

It is often necessary to combine integer and floating-point numbers in the same expression. Java permits this; however, the following two points must be considered:

- Operations between an `int` and an `int` always yeild an `int` (the remainder is lost).
- When an `int` and a `double` are mixed in an expression, the `int` is "promoted" to a `double` and the result is a `double.`

Promotion is the act of automatically converting a "lower" type to a "higher" type. The program `Hours.java` illustrates promotion by mixing `int` and `double` variables in expressions (see Figure 5).

```
 1  public class Hours
 2  {
 3     public static void main(String[] args)
 4     {
 5        int seconds = 1000000; // one million
 6        int factorInt = 3600;
 7        double factorDouble = 3600.0;
 8
 9        double hours = seconds / factorInt;
10        System.out.println(hours);
11
12        hours = seconds / factorDouble;
13        System.out.println(hours);
14     }
15  }
```

Figure 5. `Hours.java`

This program computes the number of hours in one million seconds, and generates the following output:

```
277.0
277.77777777777777
```

The result is computed twice using the same values, but the answers differ. The first answer is computed in line 9, and printed in line 10. The expression `seconds / factorInt` divides two integers, so the result is an integer and the remainder is lost. Even though the result is assigned to a `double` variable named `hours`, the division takes place first, so the remainder is lost before the assignment takes place. Since the value printed is a `double`, however, the output shows "`.0`" as the decimal portion of the value.

The second answer is computed in line 12 and printed in line 13. The expression `seconds / factorDouble` divides an integer by a `double`. The integer is promoted to a `double` before the expression is evaluated. The result is a `double` containing the usual 15, or so, digits of precision.

Note in line 7, a `double` variable `factorDouble` is declared and initialized with the constant `3600.0`. Strictly speaking, "`.0`" is not needed because the value `3600` (an `int` constant) would be promoted to a `double` anyway, as part of the assignment. It is good, however, to "show your intention" by appending "`.0`" to a whole number that is intended as a `double`. You can get into trouble by ignoring this practice when expressions involve two or more constants (as opposed to variables). For example, the expression `5 / 4` equals 1 (an `int`), whereas the expression `5 / 4.0` equals 1.25 (a `double`).

For simple assignment, an `int` can be assigned to a `double` (because of promotion); however, a `double` cannot be assigned to an `int`. In the following statements

```
int    x = 5
double y = 6.7;
y = x;            // OK! int is promoted to double
x = y;            // Wrong! can't assign a double to an int
```

the last line will cause a compiler error. If it is absolutely necessary to assign or convert an `int` to a `double`, "casting" is the answer. This is discussed in the next section.

The cast Operator

The *cast* operator explicitly converts one data type to another. Casting is useful, for example, where a conversion is necessary that does not automatically occur through promotion. Casting is also used to convert one object type to another, as we will meet later. The syntax for the cast operator is

```
(type)variable
```

or

```
(type)constant
```

The cast operator consists of the name of a data type enclosed in parentheses followed by a variable or constant. As a simple example, the following statements

```
double x = 3.14;
int y = (int)x;
System.out.println(y);
```

print `3` on the standard output. In the second line, the value of a `double` variable `x` is assigned to the `int` variable `y`. This is permitted because the `double` is cast to an `int` by preceding `x` with "`(int)`". Note that a floating-point number is truncated (not rounded) when cast to an `int`.

At this point, you might be asking yourself, "Why is the conversion from one data type to another automatic in some cases, but only allowed through casting in other cases?" Anytime there is a potential "loss of information", casting is required. The cast operator is like a safety check; it is your way of telling the compiler that you're aware of the possible consequences of the conversion.

Let's explore casting and promotion in more detail through an example program. Figure 6 contains the listing for `DemoPromoteAndCast.java`.

```
 1  public class DemoPromoteAndCast
 2  {
 3     public static void main(String[] args)
 4     {
 5        // integer promotion
 6        byte  a = 1;
 7        short b = a;           // byte promotes to short
 8        int   c = b;           // short promotes to int
 9        long  d = c;           // int promotes to long
10
11        // integer casting
12        c = (int)d;            // long casted to int
13        b = (short)c;          // int casted to short
14        a = (byte)b;           // short casted to byte
15
16        // floating-point promotion
17        float  x = 1.0f;       // 'f' needed to indicate float
18        double y = x;          // float promotes to double
19
20        // character example
21        char aa = 'Q';
22        int  xx = aa;          // character promotes to int
23        char bb = (char)xx;    // int casted to char
24
25        // floating-point casting
26        x = (float)y;          // double casted to float
27        c = (int)y;            // double casted to int
28     }
29  }
```

Figure 6. `DemoPromoteAndCast.java`

This program is as dull as they get. It receives no input and it generates no output. The key point is that it compiles without errors. All statements are perfectly legal Java statements. The objective is to exercise the rules of the Java language with respect to promotion and casting.

The program is divided into five sections. In lines 5-9, integer promotion is demonstrated, working from "lowest" to "highest". A byte variable, `a`, is declared in line 6 and assigned the value `1`. In line 7, `a` is assigned to the `short` variable `b`. Since a `byte` is 8 bits and a `short` is 16 bits, a `byte` is considered a "lower" form of an integer. The assignment is legal because the lower form is automatically promoted to the higher form. A similar process is demonstrated in line 8 (promoting a `short` to an `int`) and line 9 (promoting an `int` to a `long`).

Integer casting is demonstrated in lines 11-14. Now we are working down the hierarchy of integer types, and this requires casting. Assigning a `long` to an `int` is permitted only if the `long` is cast to an `int`, as demonstrated in line 12. Similarly, an `int` can be assigned to a `short` by casting (line 13), and a `short` can be assigned to a `byte` by casting (line 14).

Java contains only two variations of floating-point numbers, `float` and `double`, and `float` (32 bits) is "lower" than `double` (64 bits). Assigning a `float` to a `double` is allowed (line 18), as automatic promotion occurs. In line 17, note that the constant assigned to the `float` variable `x` is coded as `1.0f`. A trailing `'f'` or `'F'` indicates a floating-point number is intended as a `float`. Appending `'d'` or `'D'` indicates `double`, but this is never needed since `double` is the default for floating-point numbers.

A character example appears in lines 20-23. A `char` variable `aa` is declared and assigned the constant `'Q'`. In line 23, the `char` variable `aa` is assigned to the `int` variable `xx`. This is permissible as the `char` is automatically promoted to an `int`. The value assigned is the Unicode value of the character.

Finally, floating-point casting is demonstrated in lines 25-27. First, a `double` is assigned to a `float` (line 26). Since `float` is the lower of the two types, casting is required. Finally, a `double` is assigned to an `int` through casting.

Increment, Decrement, Prefix, Postfix

Adding or subtracting one (1) is so common in Java and other programming languages that a shorthand notation has evolved. For simple increment by 1 the `++` operator is used, and for simple decrement by 1, the `--` operator is used. So, the following two lines have the same effect:

```
x = x + 1;
x++;
```

as do

```
x = x - 1;
x--;
```

When these operators appear after the variable, as above, they are known as *postfix* operators. They can also appear in front of a variable as *prefix* operators. Although in both cases the effect is to increment or decrement the variable, there is a small but important difference. This difference is illustrated as follows:

```
int x = 5;
int y = 3 + x++;
```

After these statements execute, `y` equals 8 and `x` equals 6. Look carefully and see if you agree. Here's the operation: If a variable in an expression is bound with a postfix increment operator, the value used is the original value of the variable. The variable is incremented *after* the value is used in the expression. A similar statement can be made for the decrement operator. For these operators in the prefix position, the increment or decrement occurs *before* the variable is used in the expression. Prefix and postfix operators are summarized in Table 2.

Table 2. Prefix and Postfix Operators

Expression	Effect	Value of Expression
`x++`	increment `x` by 1	`x`
`++x`	increment `x` by 1	`x + 1`
`x--`	decrement `x` by 1	`x`
`--x`	decrement `x` by 1	`x - 1`

Operation-Assignment Shorthand

A popular shorthand notation has evolved for operators that normally take two arguments — one on the left, one on the right — such as the arithmetic operators. When these operators are combined with assignment, they can be coded using *op*= notation. For example, the following two statements have the same effect:

```
x = x + 3;
x += 3;
```

as do

```
z = z / y;
z /= y;
```

The *op*= notation is also valid for logical operators and bitwise operators, which we will meet soon.

Relational Expressions

We have used the term *expression* quite a bit in the preceding notes. For the most part, the idea is straightforward. An expression is simply one or more variables and/or constants joined by operators. An expression is evaluated and produces a result. The result of all expressions thus far was either an integer value or a floating-point value.

But, an expression can also yield a *boolean*, a result that is either `true` or `false`. Such an expression is called a *relational expression*. The result reflects how something "relates to" something else. For example, "Is the value of `x` greater than the value of `y`?" Note that the preceding poses a question. Relational expressions are usually intended to answer yes/no, or true/false, questions. Obviously, `boolean` values and `boolean` variables play an important role in relational expressions.

Relational Operators

To build relational expressions, two types of operators are used: *relational operators* and *logical operators*. Let's deal first with the relational operators. There are six relational operators, four of equal precedence:

`>` *greater than*
`>=` *greater than or equal to*
`<` *less than*
`<=` *less than or equal to*

and two just below these in precedence:

`==` *equal to*
`!=` *not equal to*

These operators, like the arithmetic operators met earlier, are sometimes called *binary* operators, because they take two arguments — one on each side. The arguments are generally integers or floating-point variables or constants. The result is a `boolean`. The following are examples of relational expressions built from relational operators: (the result is also shown)

```
3 < 4              true
7.6 <= 9           true
4 == 7             false
8.3 != 2.1         true
```

In the second line, a `double` is compared with an `int`. The `int` is promoted to `double` before the expression is evaluated.

So, what can you do with a relational expression? Since a relational expression yields either `true` or `false`, the result can be assigned to a `boolean` variable. Furthermore, a `boolean` variable, like an `int` or `double`, can be printed. So, the following two statements

```
boolean b = 8.3 != 2.1;
System.out.println(b);
```

print "`true`" on the standard output. A much more common use of relational expressions is to control program flow in `if` statements or in `while`, `do`/`while`, or `for` loops. We will meet these later.

Of course, relational expressions can also employ variables, as in the following sequence of Java statements:

```
int x = 3;
int y = 4;
boolean b = x > y;
System.out.println(b);
```

In the third line, the relational expression `x > y` is evaluated and the result, `false`, is assigned to the `boolean` variable `b`. The fourth statement prints "`false`".

Logical Operators

There are three logical operators:

&&	AND	(true if both arguments are true, false otherwise)
||	OR	(true if either argument is true, false otherwise)
!	NOT	(true if argument is false, false otherwise)

The descriptions in parentheses are usually laid out in *truth tables*, as shown in Figure 1.

(a)

a	b	a && b
false	false	false
false	true	false
true	false	false
true	true	true

(b)

a	b	a || b
false	false	false
false	true	true
true	false	true
true	true	true

(c)

a	!a
false	true
true	false

Figure 1. Truth tables for logical operators. (a) AND (b) OR (c) NOT

Logical operators are similar to relational operators in that they both produce `boolean` results. However, they differ in that logical operators also use `boolean` arguments. So the following statements make sense:

```
boolean a = true;
boolean b = false;
boolean c = a && b;
```

whereas the third statement below is nonsense:

```
int a = 3;
int b = 4;
boolean c = a && b; // wrong! && requires boolean arguments
```

The following statements, however, are perfectly reasonable:

```
int a = 3;
int b = 4;
int c = 5;
int d = 6;
boolean e = a < b && c < d;
System.out.println(e);
```

and will print "`true`" on the standard output. This result is by no means obvious, until we acknowledge the precedence relationship between the relational and logical operators. The relational operators are of higher precedence than the logical AND and logical OR operators. The parentheses in the following statement illustrate this (although they are not necessary):

```
boolean e = (a < b) && (c < d);
```

The expression `a < b` is `true` (because 3 is less than 4) and the expression `c < d` is `true` (because 5 is less than 6). The `&&` (AND) operator yields `true` if both operands are `true`, which they are in this example. So, the expression `(a < b) && (c < d)` is `true`, and this result is assigned to the `boolean` variable `e`.

The `&&` (AND) and `||` (OR) operators are binary operators, because they take two arguments. The `!` (NOT) operator is a *unary* operator, because it takes one argument. It returns the logical complement of its argument. So, the following statements

```
boolean b = true;
System.out.println(!b);
```

print "`false`" on the standard output. Note the `!` operator does not change the value of the `boolean` variable `b`; it simply returns a value which is the logical complement of `b`. Unary operators, in general, bind very tightly to their arguments; and the unary `!` operator is no exception. It is of higher precedence than all the logical or relational operators.

Lazy Evaluation

Logical operators exhibit a behaviour known as *lazy evaluation*. The evaluation of an expression with logical operators "stops" as soon as the outcome is certain. Consider the following statements:

```
int a = 3;
int b = 4;
int c = 5;
int d = 6;
boolean result = a == b && c < d;
```

The expressions in the last statement would normally be evaluated in the following order:

```
1st:  a == b
2nd:  c < d
3rd:  a == b && c < d
4th:  result = a == b && c < d
```

However, the 2nd and 3rd steps are not needed. In the first step, `a == b` yields `false` because `a` (3) does not equal `b` (4). It is not necessary to evaluate the expression `c < d`, because the outcome of the `&&` (AND) operation is now certain. It is certain because "`false` && *anything*" equals `false`. Review the truth table in Figure 1a, if you need to convince yourself of this. The expressions in the last statement are evaluated as follows:

```
1st:  a == b            (the answer is false)
2nd:  result = false;   (done!)
```

Lazy evaluation can cause subtle side effects if it is not properly understood and accounted for in the design of Java programs. It can also be used to advantage to implement safeguards. For example, the following expression

```
(z != 0) && (x / z <= y / z)
```

uses lazy evaluation to ensure a divide-by-zero error does not occur. (Parentheses are shown to clarify the operation, but they are not needed.) If `z` equals zero the first expression yields `false` and, due to lazy evaluation, the second expression is not evaluated. If `z` does not equal zero, the

first expression yields `true` and, therefore, the second expression is evaluated. In this case, the expression `y / z` can be safely evaluated because `z` does equal zero.

Lazy evaluation also occurs for the `||` (OR) operator. Consider the following relational expression:

```
rel_exp1 || rel_exp2
```

If `rel_exp1` is `true`, `rel_exp2` is not evaluated because the final result is certain: it is `true`. For a review of the `||` operator, see Figure 1b.

More examples of lazy evaluation will be given later.

Keyboard Input

Our programs thus far generated data internally, operated on the data, and printed results. A more interesting approach is to engage the user to provide input data on the host system's keyboard. In this section we will show how to receive input from the keyboard and process that input within a Java program.

The most common user interface devices are the CRT display for output and the keyboard for input. Just as `System.out` was used to output data on the host system's CRT display, `System.in` is used to receive input from the keyboard. To do this, however, the following initialization is required:

```
BufferedReader stdin =
    new BufferedReader(new InputStreamReader(System.in), 1);
```

This statement sets-up `System.in` as a buffered character *input stream*.[1] It's a bit messy, but there are two important services provided. The first is to convert raw bytes arriving from the keyboard into characters. This service is performed by the `InputStreamReader` class. Second, the characters are buffered to provide efficient reading from the input stream. This service is performed by the `BufferedReader` class. The statement declares and instantiates an object named `stdin` of the `BufferedReader` class. This is analogous to earlier statements that declared and initialized variables. Now, however, instead of a primitive data type, we have a class (`BufferedReader`) and instead of a variable, we have an object (`stdin`). So an object is "kind of like" a variable, and a class is "kind of like" a data type. Let's leave it at that for the moment and proceed with the job of inputting data from the keyboard.

Following the above statement, a line of text is read from the keyboard as follows:

```
String line = stdin.readLine();
```

Think of `stdin.readLine()` as an expression, which it is. As an expression, it is evaluated and it yields a value. This value is assigned to the variable `line`. More precisely, the `readLine()` method is called on the `stdin` object — the keyboard. A string is returned and assigned to the `String` variable `line`. The string contains the characters of a line of text entered on the keyboard.

A line of input ends when the user presses the Enter key. The Enter key generates an end-of-line code, but this is not included in the string returned by the `readLine()` method.

We'll meet strings more formally in the next section; however, for the moment, let's see what we can do with the string `line` returned by `readLine()`. Of course, the string can be printed:

```
System.out.println(line);
```

If we want the user to enter a number on the keyboard, then we must once again confront the problem of converting one type to another. For example, if the user types `3`, followed by `5`, followed by the Enter key, then the `String` variable `line` contains the character `'3'` and the

---

[1] The argument "`1`" is used in the `BufferedReader` constructor to set the buffer size to one. According to the Java API documentation, this is not needed. However, a bug in the behaviour of Java implementation on *Windows 95* and *Windows 98* causes inconsistent results with keyboard input. The initialization shown here is suggested by Sun Microsystems to correct the problem.

character '5', as opposed to the integer 35. So before the value entered can be operated on as an integer, the string must be convert to an int.

Let's focus our discussion in the context of a simple demo program. Figure 1 shows the listing for a program that prompts the user to enter a name, age, and the radius of circle.

```
 1  import java.io.*;
 2
 3  public class DemoKeyboardInput
 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        // get input from the keyboard
12        System.out.print("Please enter your name: ");
13        String name = stdin.readLine();
14
15        System.out.print("Please enter your age: ");
16        String s1 = stdin.readLine();
17
18        System.out.print("Please enter the radius of a circle: ");
19        String s2 = stdin.readLine();
20
21        // perform conversions on input strings
22        int age = Integer.parseInt(s1);          // string to int
23        double radius = Double.parseDouble(s2);  // string to double
24
25        // operate on data
26        age++;                                   // increment age
27        double area = Math.PI * radius * radius; // compute circle area
28
29        // output results
30        System.out.println("Hello " + name);
31        System.out.println("On your next birthday, you will be "
32           + age + " years old");
33        System.out.println("Area of circle is " + area);
34     }
35  }
```

Figure 1. DemoKeyboardInput.java

A sample dialogue with DemoKeyboardInput follows: (User input is underlined.)

```
PROMPT>java DemoKeyboardInput
Please enter your name: Andrew
Please enter your age: 4
Please enter the radius of a circle: 6.7
Hello Andrew
On your next birthday, you will be 5 years old
Area of circle is 141.02609421964584
```

In line 1, an import statement is necessary because the BufferedReader class and the InputStreamReader class are part the java.io package. The import statement informs the compiler of the location of these classes. We will discuss the import statement in more detail later.

In line 5, the `main()` method is defined with the `throws IOException` clause. This is required because certain types of errors, known as *exceptions*, are possible with keyboard input. A `throws` clause informs the compiler that we are aware of the possibility of such errors and that we will deal with them in an appropriate manner. Input/output exceptions are discussed in more detail later.

In lines 8-9, a link is made between `System.in` and an object named `stdin`. The effect is to setup the keyboard as an input stream for buffered character input, as discussed earlier.

With these preparations, we are ready to receive input from the keyboard. In lines 11-19, three prompts are output to the standard output, and after each the `readLine()` method is called on the `stdin` object to read a line of input from the keyboard buffer and assign it to a string variable. The first input is the user's name. It is returned as a string, and no further conversion is necessary.

The second input is the user's age. It is also input as a string; however, we wish to treat age as an integer and perform an operation on it. Since it is meaningless to perform arithmetic operations on strings, the `String` variable `s1` (line 16) must be converted to an integer. The conversion takes place in line 22 through the expression `Integer.parseInt(s1)`. This expression calls the `parseInt()` method of the `Integer` wrapper class to convert the string `s1` to an `int`. We'll say more about the `Integer` wrapper class later.

The third input is the radius of a circle. We want to treat this value as a `double` and perform operations with it. Once again, conversion is necessary. This occurs in line 23 using the expression `Double.parseDouble(s2)`. The conversion is very similar to that for an `int` except the `parseDouble()` method of the `Double` wrapper class is applied to convert the string `s2` to a `double`.

Operations are performed on the `int` variable `age` in line 26 and on the `double` variable `radius` in line 27. A reasonably precise constant representing pi is available in Java's `Math` class using the expression `Math.PI`. As with all `double` values, it is accurate with about 15 digits of precision. The results are output in lines 30-33.

Note that the `+` operator, as it appears in lines 30-33, does not represent addition. When an argument on either side of the `+` operator is a `String` variable, the operation is *concatenation* — the joining of two strings. This is an example of *operator overloading*, or the ability in an operator to perform different tasks depending on the context. The concatenation operator is discussed in more detail in the next section on strings.

Another demo program using keyboard input is listed in Figure 2. The program `HeightConversion` prompts the user to enter a height in feet and inches. The output is the height in centimeters.

```
 1  import java.io.*;
 2
 3  public class HeightConversion
 4  {
 5     private static final int INCHES_PER_FOOT = 12;  // inches per foot
 6     private static final double FACTOR = 2.564;     // cm per inch
 7
 8     public static void main(String[] args) throws IOException
 9     {
10        BufferedReader stdin =
11           new BufferedReader(new InputStreamReader(System.in), 1);
12
13        System.out.println("Please enter your height in feet and inches");
14
15        System.out.print("Feet: ");
16        int feet = Integer.parseInt(stdin.readLine());
17
18        System.out.print("Inches: ");
19        double inches = Double.parseDouble(stdin.readLine());
20
21        double cm = (feet * INCHES_PER_FOOT + inches) * FACTOR;
22
23        System.out.print("Height = " + cm + " cm");
24     }
25  }
```

Figure 2. `HeightConversion.java`

A sample dialogue follows:

```
PROMPT>java HeightConversion
Please enter your height in feet and inches
Feet: 5
Inches: 7.5
Height = 173.07 cm
```

This program includes a few new features. In line 16, the user's height is input and converted to an integer all in the same statement. The call to the `readLine()` method is simply an expression that returns a string.[2] Therefore, it is perfectly reasonable to embed this as an argument to the `parseInt()` method of the `Integer` wrapper class. In line 19, it is anticipated that the user might enter a real number, so `inches` is declared a `double` and the `parseDouble()` method is used for the conversion.

The conversion to centimeters is performed in line 21. Note the use of defined constants (lines 5-6) to make the algorithm more understandable. Integer promotion is taking place, so extra caution is warranted. The expression `feet * INCHES_PER_FOOT` is evaluated first; it involves two `int` variables so the result is an `int`. Then, this result is added to the `double` variable `inches`. The result is a `double`, and this is multiplied by the `double` variable

[2] Strictly speaking, it is incorrect to say that the `readLine()` method "returns a string". A string is an object in Java and the `readLine()` method, in fact, "returns a reference to a `String` object". Seasoned Java programmers accept the more terse "returns a string", and it appears frequently in Java textbooks and in the Java API documentation. Readers just learning the distinction between primitive data types and objects can be confused, however. And so, the point is made here to clarify the interpretation.

FACTOR to yield the final result, also a double. This is assigned to the double variable cm, which is printed in line 23.

Precedence of Operators

This is a good time to re-visit a topic discussed earlier, operator precedence. Table 1 lists all the operators in Java. Most of the operators in the table have been discussed previously. The others will be presented later on an as-needed basis. Each row represents a different precedence level, with operators farther up the table having higher precedence than operators farther down.

Table 1. Precedence of Operators (complete)

Precedence	Operator(s)	Operation
highest	`[] . () expr++ expr--`	postfix
	`++expr --expr +expr -expr`	unary
	`new (type)expr`	creation or cast
	`* / %`	multiplicative
	`+ -`	additive
	`<< >> >>>`	shift
	`> < >= <= instanceof`	relational
	`== !=`	equality
	`&`	bitwise AND
	`^`	bitwise exclusive OR
	`|`	bitwise inclusive OR
	`&&`	logical AND
	`||`	logical OR
	`?:`	conditional
lowest	`= op=`	assignment

All binary operators — those receiving two arguments — are *left-associative*, meaning they are executed left-to-right. In other words,

```
4 + 5 – 6 + 7
```

is the same as

```
((4 + 5) – 6) + 7
```

Of course, if binary operators from different rows in Table 1 are mixed in an expression, then the position in the table determines the order of evaluation. So,

```
5 – 6 * 3
```

is the same as

```
5 – (6 * 3)
```

Note that the logical AND operator (`&&`) is of higher precedence than the logical OR (`||`) operator. So,

```
a && b || c && d
```

is the same as

```
(a && b) || (c && d)
```

Although parentheses can always be added for clarity, or "just to make sure", try to avoid excessive use. Often the result is *less* clarity. It's a good idea to gain familiarity with operator precedence, and to use parentheses sparingly — only when necessary to override the natural precedence of operators.

The assignment operator (=) is *right-associative*. So,

```
a = b = c
```

is the same as

```
a = (b = c)
```

It is common in Java to mix assignment with a boolean test, for example

```
String s;
if ((s = stdin.readLine()) != null)
   /* process line */
```

Since the assignment operator (=) is of lower precedence that the inequality operator (!=), an extra set of parentheses is needed. Reworking the above fragment as

```
String s;
if (s = stdin.readLine() != null)   // WRONG!
   /* process line */
```

results in a compiler error.

The `op=` entry along the bottom row in Table 1 implies any of

```
+= -= *= /= %= >>= <<= >>>= &= ^= |=
```

Bear in mind that operator precedence combined with associativity determines the order of evaluation. So, with an expression such as

```
a + b + c
```

the compiler first evaluates `a`, then evaluates `b`, then adds the values of `a` and `b`, then evaluates `c`, then adds the value of `c` to the previous result. Order of evaluation matters, in particular, if there are side effects of any kind. Consider the following code fragment:

```
int a = 1;
int b = 2;
int c = 3;
int d = a + b++ + c + b;
System.out.println("d = " + d);
```

Can you determine the output? We'll leave it for you to explore this.

Strings

Is a string an object or a primitive data type? This question has a clear answer — a string is an object! — however, the way strings typically appear in Java programs can lead to confusion. There are shortcuts available for strings that are not available for other objects in Java. These shortcuts are highly used and they give strings the appearance of a primitive data type. But, strings are objects and they possess all the attributes and behaviours of objects in Java.

In this section, we will formally introduce strings. We'll learn how to declare and initialize `String` objects, and we'll learn how to manipulate and access strings through operators and methods.

Since strings are objects, we are, in a sense, formally introducing the concept of objects in Java, and we'll begin by presenting "strings as objects". (String shortcuts are discussed later.) In the following paragraphs, you can replace the word "`String`" by the name of any other class in Java and the story line still makes sense. So, when we talk about "instantiating a `String` object", we could just as easily be talking about "instantiating a `BufferedReader` object" or "instantiating a `PopupMenu` object". Keep this in mind and you will be well-prepared to extend your knowledge of strings to the wider and more general concept of objects.

Note in the previous paragraphs the use of the terms "string" and "`String`". In a general sense, we'll refer to a collection of characters grouped together as a "string". When referring to the same collection of characters as a formal entity in the Java language, we'll refer them as a "`String` object" or "an object of the `String` class".

Strings as Objects

A string is collection of characters grouped together. For example, "`hello`" is a string consisting of the characters `'h'`, `'e'`, `'l'`, `'l'`, and `'o'`. Note that a character constant is enclosed in single quotes, whereas a string constant is enclosed in double quotes. In Java, a string is an object. As with primitive data types, an object doesn't exist until it is declared. The following statement declares a `String` object variable named `greeting`:

```
String greeting;
```

The first hint that we are not dealing with a primitive data type, is that the word "`String`" begins with an uppercase character `'S'`. The names of Java's primitive data types all begin with lowercase letters (e.g., `int`, `double`). Indeed, class names in Java all begin with an uppercase letter. (Note that this is a convention, rather than a rule.)

The effect of the statement above is to set aside memory for an object variable named `greeting`. However the memory set aside will not hold the characters of the `greeting` string; it will hold a *reference* to the string. This important point deserves special emphasis:

An object variable holds a reference to an object!

Since the declaration does not initialize the `greeting` string, the `greeting` reference is initialized with `null` — because it refers to "nothing". Note that `null` is a reserved word in Java. This is illustrated in Figure 1a.

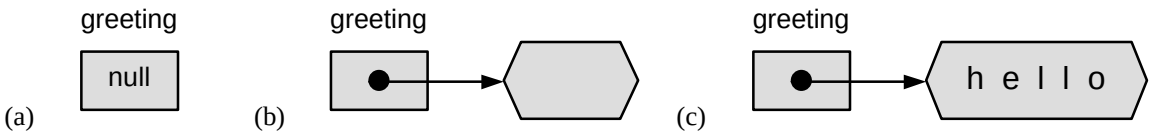


Figure 1. Creating a `String` object (a) declaration only (b) instantiating with default constructor (c) instantiating with constructor containing a string literal.

To set aside memory to hold the string, we must construct a `greeting` object. This job is performed by a type of method called a *constructor*. The process of constructing an object is called *instantiating an object*. This is performed as follows:

```
greeting = new String();
```

Declaring and instantiating a `String` object can be combined in a single statement:

```
String greeting = new String();
```

Note that the constructor method has the same name as the class. In fact, this is a rule of the Java language. The reserved word "`new`" signals to the compiler that new memory is to be allocated.

Since a constructor is a method, it is followed by a set of parentheses containing arguments passed to it. In the example above, the constructor contains no arguments. When a constructor is used without arguments, it is called a *default constructor*. In the case of the default `String` constructor, memory is set aside for the content of the string, but there are no characters in the string. This is an *empty string*, as shown in Figure 1b. The arrow emphasizes that `greeting` holds a reference, not a value. It points to the object to which `greeting` refers.

A more common way to construct a `String` object is to pass a literal — a string constant —as an argument to the constructor:

```
String greeting = new String("hello");
```

The statement above fulfills the complete task of instantiating a `String` object variable named `greeting` and initializing it with a reference to a `String` object containing "`hello`". This is illustrated in Figure 1c. Note in Figure 1, the use of a rectangle to denote a variable (in this case an object variable) and a hexagon to denote an object.

So, now that we have a `String` object variable and a `String` object, what can we do with them? Of course, the string can be printed. The statement

```
System.out.println(greeting);
```

prints "`hello`" on the standard output.

Perhaps the most fundamental of all Java operators is assignment. One string can be assigned to another, just as one integer can be assigned to another integer. Consider the following Java statements:

```
// assignment of integers
int x;
int y;
x = 5;
y = x;

// assignment of strings
String s1;
String s2;
s1 = new String("Java is fun!");
s2 = s1;
```

If you followed the preceding discussion carefully, it is apparent that these two sets of statements are slightly different. Although two `String` object variables are declared, only one `String` object is instantiated. The assignment in the final line above makes a copy of a reference to a `String` object. It does not copy the object! This is illustrated in Figure 2.

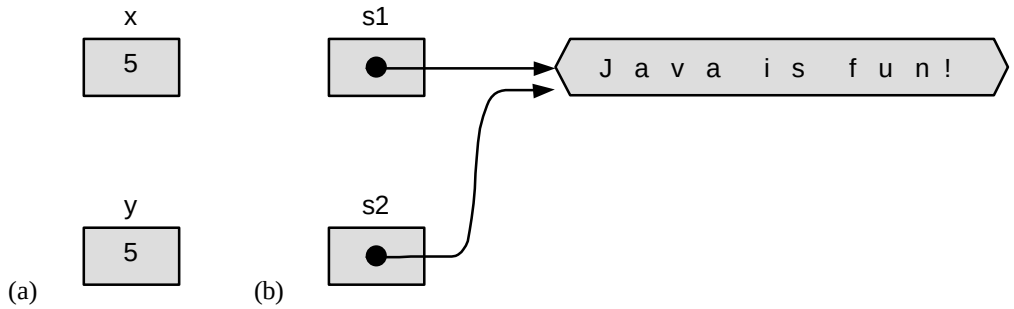


Figure 2. Assignment of (a) integers (b) strings

This example underscores a key difference between an object and a primitive data type. A primitive data type is *stored by value*, whereas an object is *stored by reference*. Stated another way, a primitive data type variable holds a value, whereas an object variable holds a reference to an object.

If you really want to make a copy of a `String` object, you can do so as follows:

```
String s1 = new String("Java is fun!");
String s2 = new String(s1);
```

Clearly, two `String` objects have been instantiated, because the constructor `String()` appears twice. In the second line the argument passed to the constructor is `s1`. The effect is to instantiate a new `String` object and initialize it with a copy of the data in `s1`. This is illustrated in Figure 3.

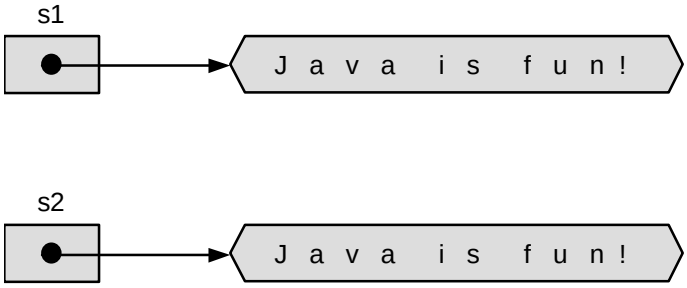


Figure 3. Copy `String` objects

Strings as Primitive Data Types

Next to numbers, strings are the most common type of data manipulated in computer programs. Because of this, `String` objects are given somewhat special treatment in the Java language. A set of shortcuts or special operations exist for `String` objects that do not exist for most other objects. These shortcuts are extremely handy, but they can also confuse students learning to program in Java. In this section we will examine how strings are commonly treated in Java. Please bear in mind that, for the most part, these discussions do not apply to other types of objects. They are unique to `String` objects.

The following shortcut provides an alternate, simplified way to instantiate a `String` object:

```
String greeting = "hello";
```

This looks very much like declaring and initializing a primitive data type, like an `int`, but don't be fooled. The statement above is just a shortcut for

```
String greeting = new String("hello");
```

This is simple enough. Let's move on to the next shortcut. Once a primitive data type, like an `int`, is assigned a value, it can be assigned a new value, and this new value replaces the old one. Similarly, once a `String` object has been instantiated and assigned a value, it can be assigned a new value, and the new value replaces the old value. Well, sort of. Let's explore this idea with a demo program. The program `DemoStringRedundancy.java` is shown in Figure 4.

```
 1  public class DemoStringRedundancy
 2  {
 3     public static void main(String[] args)
 4     {
 5        // change the value of an int variable
 6        int x = 5;
 7        x = 6;
 8        System.out.println(x);
 9
10        // change the value of a String object
11        String greeting = "hello";
12        greeting = "bonjour";
13        System.out.println(greeting);
14     }
15  }
```

Figure 4. `DemoStringRedundancy.java`

The output from this program is as expected:

```
6
bonjour
```

The statements in `DemoStringRredundancy` appear innocent enough, but something important is taking place. Replacing the value of an `int` variable with a new value does not require new memory. The new value overwrites the old value and that's the end of it. This is illustrated in Figure 5.

Figure 5. Memory allocation in `DemoStringRedundancy`
(a) after line 6 (b) after line 7

This is not the case with strings. When a `String` object is assigned a new value, the new value often differs from the old value. The string "`hello`" contains five characters, whereas the string "`bonjour`" contains seven characters. So, space for an extra two characters is needed. Where does this space come from? Let's answer this question by first presenting the following rule for `String` objects in Java:

A String object is immutable.

By "immutable", we mean that a `String` object, once instantiated, cannot change. We can perform all kinds on "read" operations on a `String` object, like determine its size or count the number of vowels, but we cannot change the content of a `String` object. The statements in `DemoStringRedundancy` are perfectly legal, and they certainly give the impression that the content of the `String` object `greeting` is changed from "`hello`" to "`bonjour`". But there is more taking place than meets the eye. All the relevant action for this discussion is in line 12. The effect of line 12 is this:

1. A new `String` object is instantiated with the `String` literal "`bonjour`".
2. The object variable `greeting` is assigned a reference to the new `String` object. (The old reference is overwritten.)
3. The old `String` object containing "`hello`" is redundant.[1]

The old `String` object is redundant because there is no longer an object variable referencing it. This is illustrated in Figure 6.

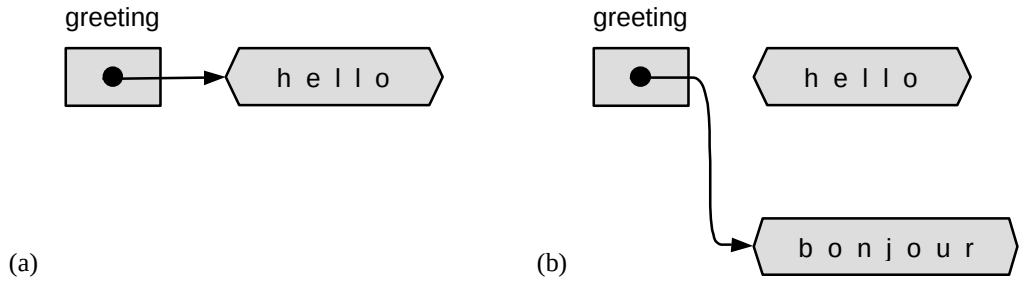


Figure 6. Memory allocation in `DemoStringRedundancy` (a) after line 11 (b) after line 12

[1] Because the original "hello" string appeared as a literal, it continues to occupy space as the program executes. Often, strings are created during the execution of a program. When such strings become redundant, the spaced occupied is eventually collected by a background process known as "garbage collection". This important service ensures that vast amounts of space are not vacated and lost as a program executes. Garbage collection occurs automatically by the method `gc()` in Java's `Runtime` class.

String Concatenation

Next to assignment, one of the most common string operations is joining two strings to form a new, larger string. This operation is called *concatenation*. Once again, `String` objects are given somewhat special treatment in Java. Strings are most-commonly concatenated using the + operator as a shorthand notation.[2] For example, the expression

```
"happy" + "birthday"
```

concatenates two strings to form a new string. The string "`birthday`" is appended to the end of the string "`happy`" to form "`happybirthday`". Forgetting to consider spaces is a common programming slip when concatenating strings. The following expression

```
"happy" + " " + "birthday"
```

gives the intended result, concatenating three string to form a new string containing "`happy birthday`".

Concatenation can involve string literals as well as `String` object variables and assignment. The statements

```
String s1 = "happy";
String s2 = " ";
String s3 = "birthday";
String name = "Billy";
String greeting = s1 + s2 + s3 + s2 + name;
```

declare and instantiate five `String` object variables and five `String` objects. The last line declares a `String` object variable named `greeting` and initializes it with a reference to a `String` object containing "`happy birthday Billy`".

If one of the arguments in a string concatenation expression is a string and the other is an `int` or a `double`, the numeric value is converted to a string as part of the concatenation. Thus, the following statements

```
int x = 53;
String answer = "The answer is " + x;
```

instantiate a `String` object containing "`The answer is 53`". It follows that a convenient way to convert a numeric variable to its string equivalent is by concatenating it with an empty string. For example, if `x` is an `int` variable containing 53, the expression

```
"" + x
```

yields a string containing "`53`".

The behaviour of the + operator is an example of *operator overloading*, as noted earlier. It is means "addition" when both operands are numeric variables, but it means "concatenation" when either operand is a string. In other words, the operation depends on the context. A brief example should suffice. The expression

---

[2] The string concatentation operator (`+`) is implemented through the `StringBuffer` class and its `append()` method. See the discussion of the `StringBuffer` class for more details.

```
53 + 7
```

uses the + operator for addition. Two integers are added to yield an integer result, 60. On the other hand, the following expression

```
"53" + 7
```

uses the + operator for concatenation, because one of the operands is a string. The other operand is an integer, but it is converted to a string during concatenation. The result of the expression is a string equal to "537".

String Methods

Other than assignment and concatenation, string operations usually employ methods of the `String` class. Including all variations, there are over 50 methods in the `String` class; however, we will only examine the most common. These are listed in Table 1.

Table 1. Common methods in the `String` class

Method	Purpose
`String(s)`	constructs a `String` object and initialize it with the string `s`; returns a reference to the object
`s.length()`	returns an `int` equal to the length of the string `s`
`s.substring(i, j)`	returns a `String` equal to a substring of `s` starting at index `i` and ending at index `j - 1`
`s.toUpperCase()`	returns a `String` equal to `s` with letters converted to uppercase
`s.toLowerCase()`	returns a `String` equal to `s` with letters converted to lowercase
`s.charAt(i)`	returns the `char` at index `i` in string `s`
`s1.indexOf(s2)`	returns an `int` equal to the index within `s1` of the first occurrence of substring `s2`; return `-1` if `s2` is not in `s1`
`s1.compareTo(s2)`	returns an `int` equal to the lexical difference between string `s1` and `s2`; if `s1` lexically precedes `s2`, the value returned is negative
`s1.equals(s2)`	returns a `boolean`; `true` if `s1` is the same as `s2`, `false` otherwise

Method Signatures

Methods are often summarized by their *signature*, which shows the first line of the method's definition. For example, the signature for the `compareTo()` method, as given in the documentation for the `String` class, is

```
public int compareTo(String anotherString)
```

The reserved word "`public`" is a *modifier*. It identifies the method's *visibility* in the larger context of a Java program. A `public` method can be accessed, or "called", from outside the class in which it is defined; it is visible anywhere the class is visible. (A method is declared `private` if it helps a public method achieve its goal but it is not accessible outside the class.)

The reserved word "`int`" identifies the data type of the value returned by the method. The `compareTo()` method returns an integer. A method can also return an object; so a class name may appear instead of a data type.

Next comes the name of the method. It is a Java convention that names of methods always begin with a lowercase letter, as noted earlier. If the method's name is composed of more than one word, the words are run together and subsequent words begin with an uppercase character. Hence, `compareTo()` begins with a lowercase letter; but uses uppercase "`T`" in "`To`". Constructor methods are an exception, since they always use the same name as the class, and therefore always begin with an uppercase character.

Following the method's name is a set of parentheses containing the arguments passed to the method separated by commas. Each argument is identified by two words. The first is its type; the second is its name. The name is arbitrary and need not match the name of an argument in the program that uses the method.

String Methods

The demo program `DemoStringMethods.java` exercises all the methods in Table 1. The listing is given in Figure 7.

```
 1  public class DemoStringMethods
 2  {
 3     public static void main(String[] args)
 4     {
 5        String s1 = new String("Hello, world");
 6        String s2 = "$75.35";
 7        String s3 = "cat";
 8        String s4 = "dog";
 9
10        System.out.println(s1.length());
11        System.out.println(s1.substring(7, 12));
12        System.out.println(s1.toUpperCase());
13        System.out.println(s1.toLowerCase());
14        System.out.println(s1.charAt(7));
15        System.out.println(s2.indexOf("."));
16        System.out.println(s2.substring(s2.indexOf(".") + 1, s2.length()));
17        System.out.println(s3.compareTo(s4));
18        System.out.println(s3.equals(s4));
19     }
20  }
```

Figure 7. `DemoStringMethods.java`

This program generates the following output:

```
12
world
HELLO, WORLD
hello, world
w
3
35
-1
false
```

We'll have many opportunities the meet these methods later in more interesting programs. For the moment, let's just examined the basic operation of each.

String()

In line 5, the `String()` constructor is used to instantiate a `String` object. A reference to the object is assigned to the object variable `s1`. In fact, `String` objects are rarely instantiated in this manner. It is much more common to use the shorthand notation shown in lines 6-8.

length()

In line 10, the `length()` method determines the length of `s1`. The syntax shown in line 10 illustrates an "instance method" operating on an "instance variable". Dot notation connects the instance variable (`s1`) to the instance method (`length()`). `s1` is called an instance variable because the object to which it refers is an "instance of" an object — a `String` object.

The term `"s1.length()"` is an expression. It returns an integer value equal to the length of the string. So, the term `"s1.length()"` can be used anywhere an integer can be used, such as inside the parentheses of the `println()` method. The length of `s1` is 12, as shown in the 1st line of output.

substring()

In line 11, a substring is extracted from `s1`. The substring begins at index 7 in the string and ends at index 11. In counting-out character positions within strings, the first character is at index "0". This is illustrated in Figure 8.

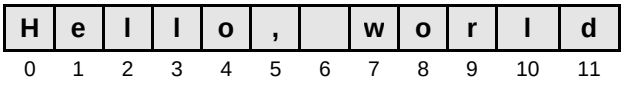


Figure 8. Characters positions within a string

Note that the second argument in the `substring()` method is the index "just passed" the last character in the substring. So, to extract the substring "`world`" from "`Hello, world`", the arguments passed to the `substring()` method are 7 and 12, respectively. The 2nd line of output shows the result: `world`.

Since the first position in a string is at index 0, the size of the string returned by the `length()` method is "one greater than" the last index in the string.

It is illegal to access a non-existent character in a string. This would occur, for example, by using the `substring()` method on an empty string:

```
String s1 = "";
String s2 = s1.substring(0, 1);
```

When these statements execute, a `StringIndexOutOfBounds` exception occurs and the program crashes with a run-time error (unless the exception is caught and handled explicitly in the program).

toUpperCase () and toLowerCase()

Line 12 demonstrates a simple conversion of a string to uppercase characters. Non-alphabetic characters are left as is; alphabetic characters are converted to uppercase. Note that the object referenced by the instance variable, `s1` in this case, remains unchanged. The method simply returns a reference to a new `String` object containing only uppercase characters. If, by chance,

the string does not contain any lowercase characters, then no conversion is necessary and a reference to the original object is returned.

Conversion to lowercase characters, shown in line 13, is similar to the process described above, except uppercase characters are converted to lowercase. The result of the case conversions is shown in the program's output in the 3rd line ("`HELLO, WORLD`") and 4th line ("`hello, world`").

charAt()

In line 14, the `charAt()` method is used to determine which character is at index 7 in the string referenced by `s1`. As evident in Figure 8, the answer is "`w`" and this is shown in the 5th line of output.

indexOf()

In line 15, the `indexOf()` method is demonstrated. The method is called on the instance variable `s2`, which references a `String` object containing `"$75.35"` (see line 6). The decimal point ("`.`") is at index 3, as shown in the 6th line of output.

Line 16 demonstrates the convoluted sorts of operations that can be performed with methods of the `String` class. The statement retrieves the decimal places from the `String` object referenced by instance variable `s2`. Have a close look at the syntax as see if you agree with the result shown in the 7th line of output.

compareTo()

In line 17 the `compareTo()` method is demonstrated. As noted in Table 1, the method performs lexical comparison between two strings and returns an integer equal to the lexical difference between the two strings. This sounds complex, but it's really quite simple. "Lexical order" is, for the most part, "dictionary order"; so "able" precedes "baker", which in turn precedes "charlie", and so on.

The lexical difference between two strings is simply the numerical difference between the Unicodes of the characters at the first position that differs in the two strings. For, example the expression `"able".compareTo("baker")` equals -1. The first characters differ and the numerical difference between them is one (1). Since `'a'` precedes `'b'` the value returned is minus one (-1). The expression `"aardvark".compareTo("able")` also equals -1, but this is the lexical difference between the second two characters (since the first two are the same). In the example program, the comparison is between "`cat`" and "`dog`" (see line 17). The lexical difference, -1, appears in the 8th line of output.

There are a couple of situations not accounted for in the discussion above. The numerical difference between two characters that differ only in case (e.g., `'a'` and `'A'`) is 32, with uppercase characters lexically preceding lowercase characters. So, for example, the expression `"zebra".compareTo("ZEPHYR")` equals 32. Comparisons that involve digits or punctuation symbols are a little trickier to evaluate. The Unicodes for these and other special symbols are given in Appendix A.

If there is no index position where the two strings differ, yet one string is shorter than the other, then the shorter string is considered to lexically precede the longer string. The value returned is the difference in the lengths of the two strings. So, for example, the expression

```
"lightbulb".compareTo("light")
```

equals 4 and

```
"light".compareTo("lightbulb")
```

equals -4.

equals()

The last string method demonstrated in Figure 7 is the `equals()` method. `equals()` is a similar to `compareTo()` except it only tests if for equality or inequality. The return type is a `boolean`: `true` if the two strings are identical, `false` otherwise. Since "cat" is not equal to "dog", the result is `false`, as seen in the 9th line output.

Note that the relational test for equality (`==`) is a valid string operator; however, its behaviour is distinctly different from the `equals()` method. For example, the expression

```
s1 == s2
```

returns `true` if `s1` and `s2` refer to the same object, where as the expression

```
s1.equals(s2)
```

returns `true` if the character sequence in `s1` is the same as in `s2`.

Chapter 3

Program Flow

Choices

The example programs so far were all similar in one respect: they all executed in a "straight line". Each statement in the source program executed after the statement on the preceding line. This is convenient if our goal is to learn the basic elements of Java programs, however the demo programs were not very interesting. By providing a capability to "make choices" or to perform simple tasks "repeatedly" or "in a loop", we can devise much more exciting programs. We can begin to solve non-trivial problems, and, more importantly, we can begin to see why the string literal "`Java is fun!`" was used so much in the preceding examples.

We will now study how to alter the straight-line flow of programs by making choices with the `if` statement and the `switch` statement, and by executing a series of statements in a loop using the `while`, `do/while`, and `for` statements. Let's begin with making choices.

Making a choice in a computer program is like confronting a fork in a road. The route taken determines the sights encountered. The choice is something like, "For a scenic trip, turn left; for a fast route, turn right". Such decisions also surface in programming language like Java. Let's begin our examination of programming choices with one of the oldest tools in computer programming: *flowcharts*. Figure 1 shows two flowcharts illustrating (a) a straight-line program and (b) a program that includes a choice.

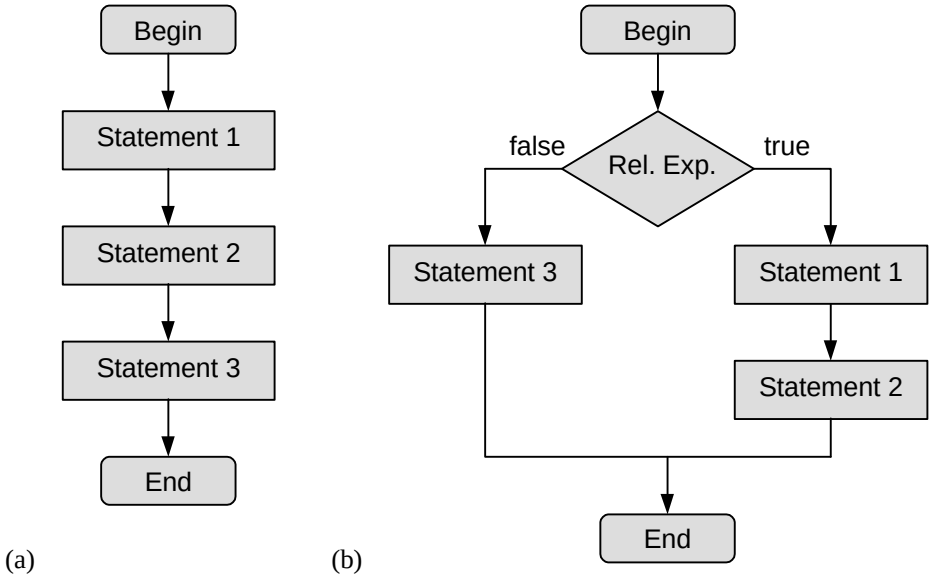


Figure 1. Program flow (a) straight-line execution (b) choice

There are detailed conventions on the design of flowcharts; however, they are not presented in this text. For our purposes, the three basic shapes shown in Figure 1 will suffice. The shapes are connected by arrows showing the flow from one program element to the next. Program termination points are depicted within rounded rectangles using the labels "begin" and "end". These refer to the terminal points for an entire program, a method, or any other subsection of code. Statements are enclosed in rectangles, and decisions are enclosed in diamonds. A decision is always based on a relational expression with `true` or `false` value.

Unfortunately, the picture in Figure 1b is awkward to convert to computer instructions. The problem is that program statements are entered line-by-line, one after the other. At some point,

the two-dimensional layout in Figure 1b must be reworked in a one-dimensional format, as shown in Figure 2.

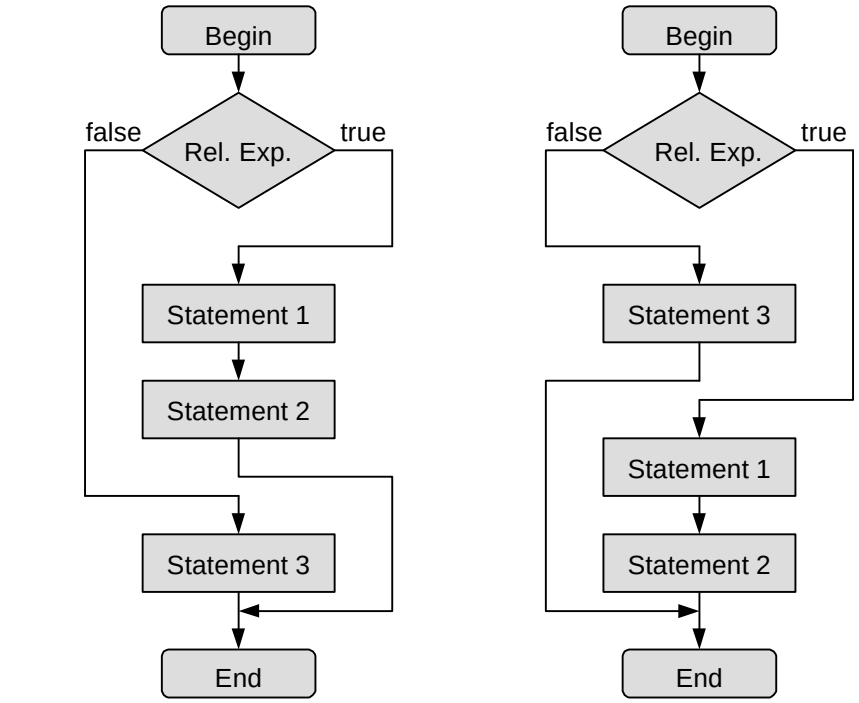


Figure 2. Straight-line flowcharts with choices

Each of part (a) and part (b) in Figure 2 is identical in flow to Figure 1b. However, the pictures are considerably less "pretty". The most useful information in Figure 2 lies in the lines and arrows, rather than in the juxtaposition of the shapes, as in Figure 1b. Figure 1b one more clearly represents a human's mental model of flow with decisions, whereas Figure 2 more clearly represents the line-by-line linear layout of program statements in a source file. The point is this: flowcharts are considerably less useful than they are often made out to be. In this author's experience, programmers rarely solve problems by first constructing a flowchart and then developing the code using the flowchart as a guide. Even if a flowchart is a requirement of an assignment in a programming course, students write and debug the code first, then construct the flowchart later to meet the requirements of the assignment. Is this a bad approach? Perhaps not. Forcing the development process along ordered, isolated activities is usually overstressed and probably wrong. As research in artificial intelligence has discovered, modeling human intelligence is a slippery business. Humans appear to approach the elements of a situation in parallel, simultaneously weighting possible actions and proceeding by intuition.

We will use flowcharts only sparingly from this point onward.

A much more useful aid to the design of programs with non-linear flow is a technique inherited from structure programming languages, such as C or Pascal. The technique is *indentation*. Indentation is, by far, the best way to reveal the flow or possible paths that a program may follow during execution. Not adhering to a consistent style of indentation is an invitation for trouble. The style shown in these notes is consistent. You may adopt it, or adopt some other style; but, be forewarned, as soon as your style of indentation wavers, you will introduce bugs that take far more time to correct than the time to indent your source code. As a guideline, make sure every line of source code has the proper indentation. If you aren't sure how much indentation is needed,

stop! Take a few moments and consider the big picture. How does this statement interact with the statements around it? Don't proceed until you are sure. Okay, so much for the sermon, let's begin our study of program flow by examining Java's `if` statement.

if Statement

In a computer program, we don't usually think about roads and which turn leads to a scenic route. We are more likely to confront situations like this: "If the user typed 'Q', quit the program, otherwise continue". Note the use of the word "if" in the preceding phrase. Not surprisingly, our study of program flow begins with Java's `if` statement.

There are two forms of the `if` statement: `if` and `if/else`. The syntax of the `if/else` form is

```
if (relational_expression)
   statement1;
else
   statement2;
statement3;
```

where "*relational_expression*" is any term or expression with a boolean value. If the relational expression is `true`, statement #1 executes, otherwise statement #2 executes. After statement #1 or #2 executes, the program continues with statement #3. Any statement can be replaced by a statement block using braces:

```
if (relational_expression)
{
   statement1a;
   statement1b;
   statement1c;
}
else
   statement2;
statement3;
```

The `else` part is optional:

```
if (relational_expression)
   statement1;
statement2;
```

With this form, statement #1 executes only if the relational expression is true. Either way, the program proceeds with statement #2. Figure 3 shows the flowcharts for an `if/else` statement and an `if` statement.

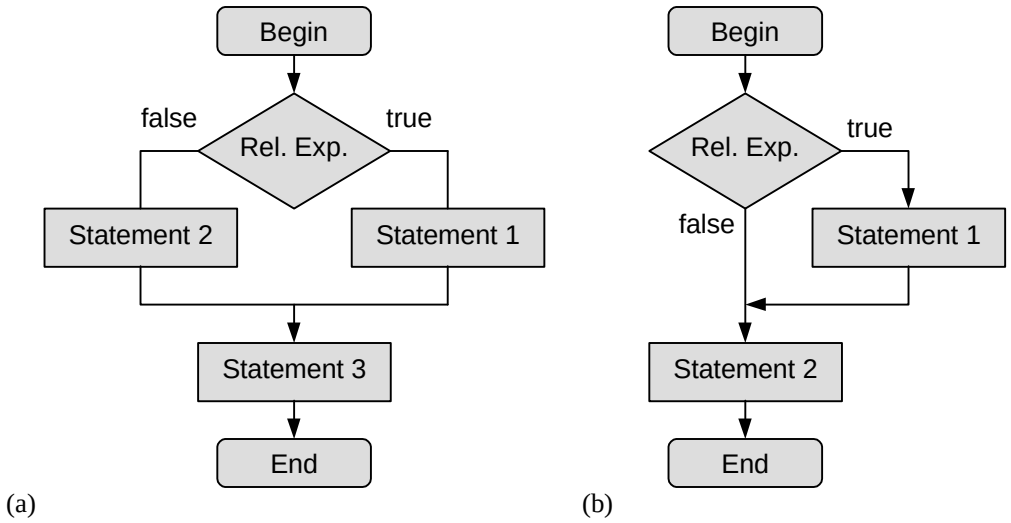


Figure 3. Flowarts (a) `if/else` statement (b) `if` statement

Let's examine the `if/else` statement in a simple demonstration program. `DemoIfElseStatement` prompts the user to enter his or her age. The outputs depends on the value entered (see Figure 4).

```
 1  import java.io.*;
 2
 3  public class DemoIfElseStatement
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        BufferedReader stdin =
 8           new BufferedReader(new InputStreamReader(System.in), 1);
 9
10        System.out.println("Welcome to the Java Roadhouse!");
11        System.out.print("Please enter your age: ");
12        int age = Integer.parseInt(stdin.readLine());
13        if (age < 19)
14           System.out.println("Here's a ticket for a free soda");
15        else
16           System.out.println("Here's a ticket for a free beer");
17     }
18  }
```

Figure 4. `DemoIfElseStatement.java`

All the elements in this program were discussed earlier, except for the `if/else` statement in lines 13-16. Two sample dialogues are shown below. (User input is underlined.)

```
PROMPT>java DemoIfElseStatement
Welcome to the Java Roadhouse!
Please enter your age: 17
Here's a ticket for a free soda
Please enter

PROMPT>java DemoIfElseStatement
Welcome to the Java Roadhouse!
Please enter your age: 23
Here's a ticket for a free beer
Please enter
```

In line 13, an `if` statement uses the relational expression "`age < 19`" to determine which of two statements to print. The variable `age` is compared with 19 using the < operator. The result is a boolean value, `true` if `age` is less than 19, `false` if `age` is 19 or greater.

In many programming situations the else part of an if statement is not necessary. For example, the following statement

```
if (temperature < 15)
   System.out.println("Don't forget your coat!");
...
```

prints a reminder to bring your coat if the variable `temperature` is less than 15. If `temperature` is 15 or greater, the reminder is not printed. Either way, the program continues with the next statement. Note above that there is no statement that executes only if the temperature is 15 or greater.

Nested `if` statements

It is quite common to have `if` statements nested inside other `if` statements to provide multiple alternatives in a programming problem. This is illustrated in Figure 5.

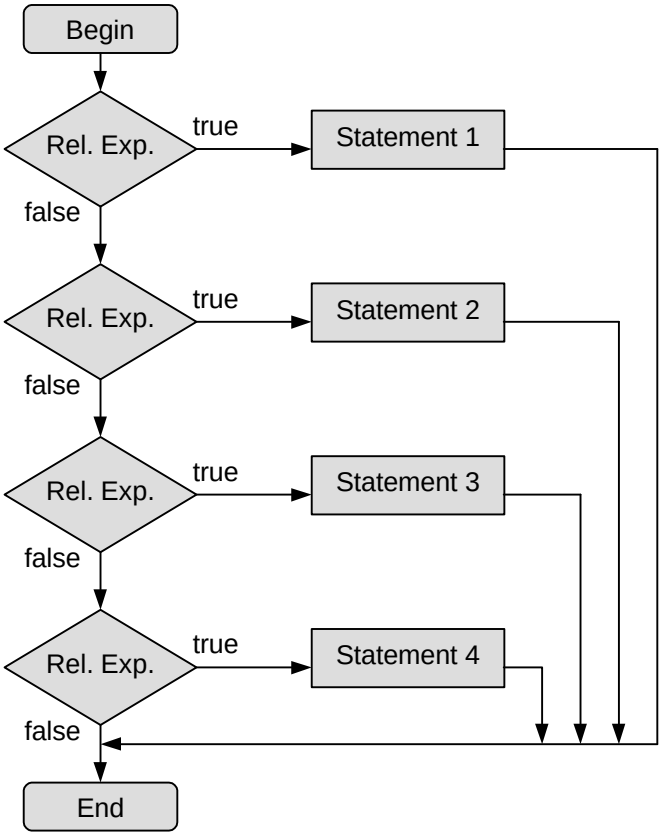


Figure 5. Nested `if` statements for multiple alternatives

An example is shown in the program `Grades.java` which outputs a letter grade based on a numeric mark.

```
 1  import java.io.*;
 2
 3  public class Grades
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        BufferedReader stdin =
 8           new BufferedReader(new InputStreamReader(System.in), 1);
 9
10        System.out.print("Enter your mark in percent: ");
11
12        double mark = Double.parseDouble(stdin.readLine());
13
14        if (mark >= 80)
15           System.out.println("You got an A");
16        else if (mark >= 70)
17           System.out.println("You got a B");
18        else if (mark >= 60)
19           System.out.println("You got a C");
20        else if (mark >= 50)
21           System.out.println("You got a D");
22        else
23           System.out.println("You got an F");
24     }
25  }
```

Figure 6. `Grades.java`

A sample dialogue with this program follows:

```
PROMPT>java Grades
Enter your mark in percent: 66.6
You got a C
```

At the point where a relational expression yields `true` in `Grades.java`, the letter grade is printed, and no further relational expressions are evaluated. Note that the last `else` handles the "none of the above" condition.

It can be difficult determining the pairings for the `if` and `else` parts of nested `if/else` statements (particularly if indentation is inconsistent). The rule is this: an `else` is associated with the nearest preceding `if` that does not have an `else` immediately following it.

Figure 7 illustrates nesting `if/else` statements within each half of a preceding `if/else` statement.

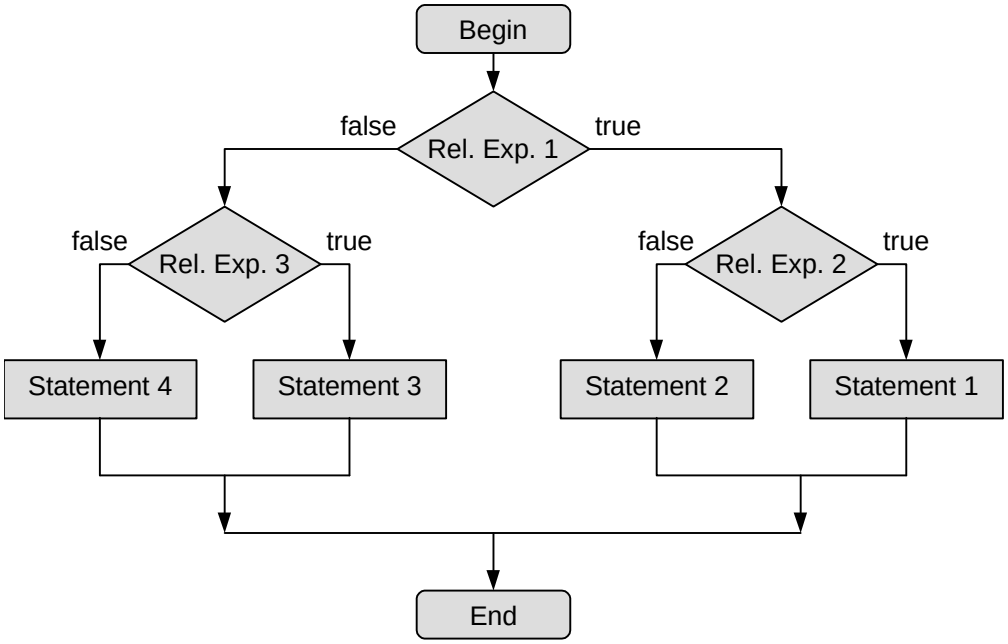


Figure 7. Nesting `if/else` inside `if/else`

Two possible implementations are given in Figure 8.

```
if (rel_exp1)
   if (rel_exp2)
      statement1;
   else
      statement2;
else
   if (rel_exp3)
      statement3;
   else
      statement4;
```

(a)

```
if (rel_exp1 && rel_exp2)
   statement1;
else if (rel_exp1 && !rel_exp2)
   statement2;
else if (!rel_exp1 && rel_exp3)
   statement3;
else if (!rel_exp1 && !rel_exp3)
   statement4;
```

(b)

Figure 8. Two implementations for the nested `if/else` statements in Figure 7.

The code in Figure 8a is clearly easier to follow; however the code in Figure 8b takes fewer lines. The solution in Figure 8b combines relational expressions to more directly arrive at the desired statement. Note the important role of the logical `!` (NOT) operator; it effectively captures the "`false`" paths through the `if/else` statements.

There are advantages and disadvantages to each approach in Figure 8. In Figure 8a, only two relational expressions are evaluated in reaching the correct statement. In Figure 8b, as many as eight relational expressions may be evaluated. So, in general the approach in Figure 8b is slower.

The approach in Figure 8b also suffers from other "possible" problems. If statement #4 is to be executed, then `rel_exp1` is evaluated four times. Not only is this time consuming, if the expression includes any operations that change the content of a variable in the expression (e.g., `x++`), then the variable will not hold the same value each time the expression is evaluated. There may be a subtle, hidden bug lieng in wait! Here's another "possible" problem with the approach in Figure 8b. Due to lazy evaluation (see below), it is not certain how many times `rel_exp2` or

rel_exp3 are evaluated. It is not possible, therefore, to predict the final value of any variable used in these expressions that changes. Again, a bug may exist that only surfaces under certain initial conditions.

Pay special attention to the effect of lazy evaluation for logical operators. In the first if expression in Figure 8b, rel_exp2 is not evaluated if rel_exp1 is false. (If rel_exp1 is false, the combined expression *must be* false!) This is not a bug; it is simply a behaviour that must be understood by Java programmers. The trick is to make sure no variables used in rel_exp2 are assigned new values!

Despite the caveats above, programmers still construct if/else hierarchies with complex relational expressions. Is this bad? Not necessarily! As noted earlier, humans approach problems and their solutions in a variety of complex and intuitive ways. To stifle such creativity by imposing rigid coding guidelines probably brings forth more problems than those guidelines are worth. The main prerequisite to proceeding creatively in solving problems in Java is a thorough understanding of the constructs of the language.

While we're on the subject of lazy evaluation, let's examine a powerful technique to incorporate safeguards in relational expressions. Suppose s references a string of characters. Suppose further that we want to do something special if the string begins with 'Z'. This could be achieved as follows:

```
if (s.substring(0, 1).equals("Z"))
   System.out.println("Begins with Z");
```

If, by chance, s references an empty string, we're in trouble because accessing a non-existent index in a string is illegal and will cause a run-time error or exception. This is effectively averted as follows:

```
if (s.length() > 0 && s.substring(0, 1).equals("Z"))
   System.out.println("Begins with Z");
```

If s references an empty string, the first relational expression above yields false. Since the && (AND) operator connects the two relational expressions, the combined result *must be* false. Therefore, due to lazy evaluation, the second expression is not evaluated and the error is averted.

Selection Operator

The *selection* operator (? :) provides a convenient shorthand notation for certain conditional assignments. For example, the following if/else statement

```
if (a < b)
   z = a;
else
   z = b;
```

can be coded as

```
z = a < b ? a : b;
```

The variable z is assigned the value of a (if a is less than b) or the value of b (if a is greater than or equal to b).

The general syntax for the ? : operator is

```
rel_exp ? exp1 : exp2
```

This is just an expression which returns the value of *exp1* (if *rel_exp* is `true`) or *exp2* (if *rel_exp* is `false`). Expression *rel_exp* is a relational expression, whereas *exp1* and *exp2* are expressions returning any data type or an object. Since the `? :` operator takes three operands, it is called a *ternary* operator.

switch Statement

As with the selection operator, the `switch` statement provides a convenient alternative (but it is not essential). Its main use is to avoid an unsightly series of `if/else` statements when choosing one alternative among a set of alternatives. The syntax for `switch` statement follows:

```
switch (integer_expression)
{
   case 0:
      statement0;
      break;
   case 1:
      statement1;
      break;
   case 2:
      statement2;
      break;
   ...
   default:
      default_statement;
}
```

Note that "switch", "case", "break", and "default" are reserved words in Java. If the integer expression equals 0, statement #0 executes, then the `break` statement executes. The `break` statement causes an immediate exit from the `switch`. If the integer expression equals 1, statement #1 executes, and so on. If no case matches the integer expression, the default statement executes. If the `break` statements are omitted, the correct case is still chosen; however, execution proceeds through the other statements until the end of the `switch` or until a `break` statement is eventually reached. This is usually a programming error. The `break` statement is discussed in more detail later.

The program `TaxRate.java` demonstrates the `switch` statement.

```
 1  import java.io.*;
 2
 3  public class TaxRate
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        BufferedReader stdin =
 8           new BufferedReader(new InputStreamReader(System.in), 1);
 9
10        System.out.println("What is your income bracket?");
11        System.out.println("1 = less than 25K");
12        System.out.println("2 = 25K to 50K");
13        System.out.println("3 = more than 50K");
14        System.out.print("Enter: ");
15        int bracket = Integer.parseInt(stdin.readLine());
16        switch (bracket)
17        {
18           case 1:
19              System.out.println("Pay no taxes");
20              break;
21           case 2:
22              System.out.println("Pay 20% taxes");
23              break;
24           case 3:
25              System.out.println("Pay 30% taxes");
26              break;
27           default:
28              System.out.println("Error: bad input");
29        }
30     }
31  }
```

Figure 9. `TaxRate.java`

A sample dialogue follows:

```
PROMPT>java TaxRate
What is your income bracket?
1. less than 25K
2. 25K to 50K
3. more than 50K
Enter: 2
Pay 20% taxes
```

This program implements a simple menu. The user is prompted to enter 1, 2, or 3 to select an income bracket. The value is inputted and converted to an `int` in line 15, then passed as an argument to the `switch` statement in line 16. In the dialogue above, the user entered "`2`", so the message printed was "`Pay 20% taxes`".

The statements in lines 16-29 could be coded using a series of `if/else` statements as follows:

```
if (bracket == 1)
   System.out.println("Pay no taxes");
else if (bracket == 2)
   System.out.println("Pay 20% taxes");
else if (bracket == 3)
   System.out.println("Pay 30% taxes");
else
   System.out.println("Error: bad input");
```

The arrangement above takes fewer lines, but it's also less clear. The difference is more dramatic is the statements are replaced by statement blocks. The choice of using a `switch` statement or a series of `if/else` statements is a personal preference, and we'll not adhere strictly to either style in these notes.

The `switch` statement only works with integer arguments. This is a limitation in some situations. Since an `if` statement uses a relational expression, it is more flexible. However, with some "preprocessing" a switch can often still be used. For example, assume a program inputs words and processes them differently depending on their length. If `word` is a `String` object variable referencing the inputted word, then the processing could proceed as follows using `if/else` statements:

```
if (word.length() == 1)
   ...
else if (word.length() == 2)
   ...
else if (word.length() == 3)
   ...
else
   ...
```

Each ellipsis (`...`) would be replaced by a statement or statement block, as appropriate. The final `else` handles the "none of the above" situation of `word` referencing an empty string. (An empty string has length zero.) The same effect is achieved using a `switch` statement as follows:

```
int len = word.length();
switch (len)
{
   case 1:    // length is 1
      ...
      break;
   case 2:    // length is 2
      ...
      break;
   case 3:    // length is 3
      ...
      break;
   default:   // empty string (length is 0)
      ...
}
```

Besides the more appealing look, the arrangement above has one other benefit. The expression "`word.length() == n`" in the preceding example is re-evaluated for each if-test. This is time consuming since it requires a call to a class method. On the other hand the integer comparison in the `switch` statement is extremely fast. The speed difference may not appear in small programs, but it may be a concern in some situations.

Let's reconsider this example with a slightly different goal. Suppose the words are processed separately depending on the first letter in the word. Can a `switch` statement be used? Yes, but a little ingenuity is required. Here's the approach. First, we initialize an `int` variable `code` either with -1 if `word` references an empty string, or with the lexical difference from `'a'` of the first letter in `word`:

```
int code;
if (word.length() == 0)
   code = -1;
else
   code = word.substring(0, 1).toLowerCase().compareTo("a");
```

So, if the word is "able", `code` is 0, if the word is "baker", `code` is 1, if the word is "charlie", `code` is 2, and so on. It is important to check for an empty string, as above. The expression `word.substring(0, 1)` generates a run-time error (the program crashes!) if `word` references an empty string. The arrangement above effectively guards against this. With this preparation, the words are processed using a `switch` statement as follows:

```
switch (code)
{
   case 0:     // word begins with 'a' or 'A'
      ...
      break;
   case 1:     // word begins with 'b' or 'B'
      ...
      break;
   case 2:     // word begins with 'c' or 'C'
      ...
      break;
   ...
   default:    // empty string or doesn't begin with a letter
      ...
}
```

Loops

Loops provide the mechanism to do simple operations repeatedly. In this section, we'll examine Java's three loops statements: the `while` loop, the `do/while` loop, and the `for` loop. Let's begin with the `while` statement.

while Loop

The syntax for a while loop is

```
while (relational_expression)
   statement;
```

It is more common to see a `while` loop with a statement block:

```
while (relational_expression)
{
   statement1;
   statement2;
   ...
}
```

A `while` statement consists of the reserved word `while` followed by a relational expression in parentheses. The relational expression is used for "loop control". Following the relational expression is a statement or statement block.

Here's how a `while` statement works: First the loop control relational expression is evaluated. If the result is `true` the statement or statement block executes. Then the relational expression is evaluated again. If the result is still `true`, the statement(s) executes again. This continues until the relational expression is evaluated and yields `false`, whereupon the statement(s) is skipped and execution continues after the statement or statement block.

If the relational expression yields `false` the first time it is evaluated, then the statement(s) never executes. If the relational expression yields `true` the first time it is evaluated, then the statement(s) executes at least once. Clearly, the relational expression must eventually yield `false`, otherwise an infinite loop results.

The flowchart for the while statement is shown in Figure 1.

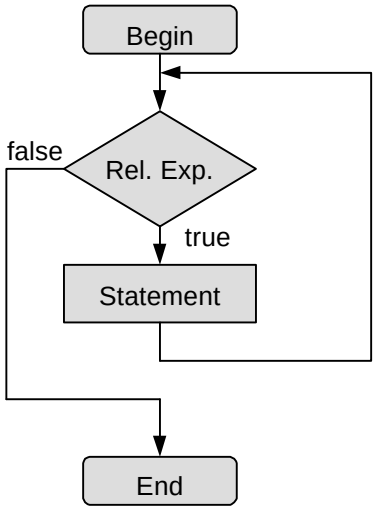


Figure 1. Flowchart for `while` statement

The program `TableOfSquares.java` outputs a table of the squares of integers from 0 to 5 using a `while` statement (see Figure 2).

```
 1  public class TableOfSquares
 2  {
 3     public static void main(String[] args)
 4     {
 5        System.out.print("===============\n"
 6                         +  " x  Square of x\n"
 7                         +  "--- -----------\n");
 8        int x = 0;
 9        while (x < 6)
10        {
11           System.out.println(" " + x + "       " + (x * x));
12           x++;
13        }
14        System.out.println("===============\n");
15     }
16  }
```

Figure 2. `TableOfSquares.java`

This program generates the following output:

```
===============
 x  Square of x
--- -----------
 0       0
 1       1
 2       4
 3       9
 4       16
 5       25
===============
```

The `int` variable `x` is used for loop control. It is declared and initialized to zero in line 8, just before the `while` loop begins. In line 9, the relational expression "`x < 6`" is evaluated. Since `x` was just initialized to zero, the result is `true`, so the statement block in lines 10-13 executes.

A print statement outputs `x` and `x * x` with the appropriate spacing for the table to look good. Then, `x` is increment by one using the postfix increment operator in line 12. Execution then moves back to the loop control expression. `x` is now 1, so the loop control expression again equals `true` and the statement block executes again. Eventually `x` is incremented to 6, and at this point the loop control expression yields `false` (6 is not less than 6!). Execution then proceeds out of the `while` loop to line 14, where the bottom rule for the table is printed.

It is, of course, the `int` variable `x` that is crucial for the while loop to achieve the desired effect. It is a general rule for loops that something in the loop must alter a variable in the loop control expression, otherwise an infinite loop occurs.

do/while Loop

The main difference between a `while` loop and a `do/while` loop is that the statement or statement block executes *at least once* with a `do/while` loop. This is illustrated in the flowchart in Figure 3.

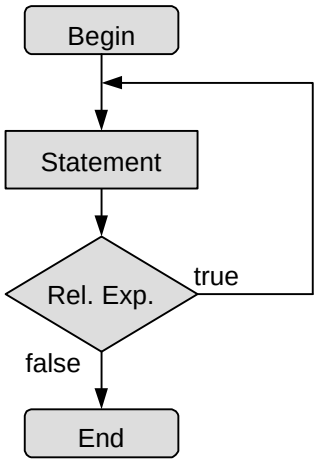


Figure 3. Flowchart for the `do/while` loop

The syntax for a `do/while` loop is a little trickier than for a `while` loop:

```
do
   statement;
while (relational_expression);
```

or

```
do
{
   statement1;
   statement2;
   ...
} while (relational_expression);
```

The loop begins with the reserved word `do`. This is followed by a statement or statement block. The loop control relational expression appears at the end of the loop in parentheses following the reserved word `while`. Note that a semicolon is required after the closing parenthesis. A `do/while` loop cannot be constructed without the closing `while`.

Figure 4 shows a program using a `do/while` loop to print the message "`Java is fun!`" five times.

```
 1  public class JavaIsFun
 2  {
 3     public static void main(String[] args)
 4     {
 5        int i = 0;
 6        do
 7        {
 8           System.out.println("Java is fun!");
 9           i++;
10        } while (i < 5);
11     }
12  }
```

Figure 4. `JavaIsFun.java`

The `int` variable `i` is declared and initialized with 0 in line 5, just before the `do/while` loop begins. Note that the statements in the `do/while` loop in lines 8 and 9 in execute *at least once*, since the loop control relational expression is not evaluated until line 10. This is the key difference between a `while` loop and a `do/while` loop and it is the primary characteristic that determines which type we choose when setting up a loop.

for Loop

A popular alternative to the `while` loop is the `for` loop. More often than not, a loop uses a *loop control variable*. This variable is *initialized* before the loop begins, it is used in the *loop control* relational expression, and it is *adjusted* at the end of the loop. When these conditions exist, a `for` loop can often do the job. This is illustrated in Figure 5.

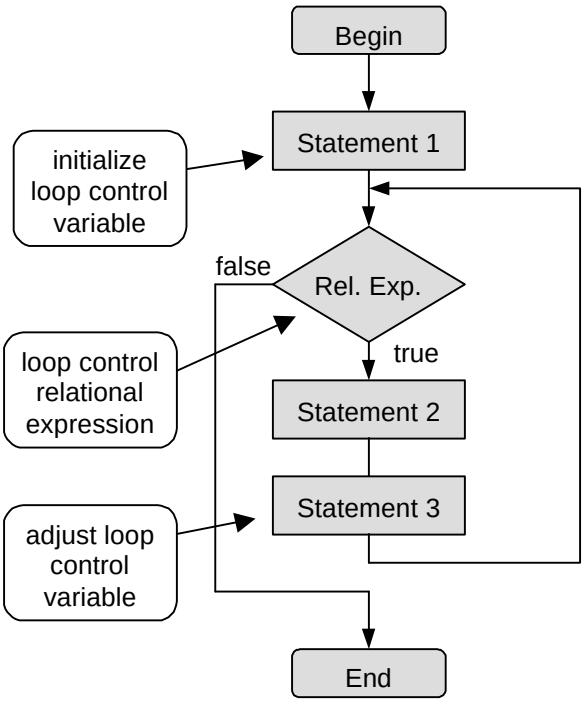


Figure 5. Flowchart for `for` loop

The general syntax for the `for` loop is

```
for (exp1; rel_exp; exp2)
   statement;
```

This is the same as

```
exp1;
while (rel_exp)
{
   statement;
   exp2;
}
```

Expression #1 and expression #2 can be any type of expression, but, most commonly, expression #1 is a statement that initializes a loop control variable, and expression #2 is a statement that adjusts the loop control variable. Within parentheses is the loop control relational expression that determines if the loop is to execute again.

The following `for` loop will print "Java is fun!" five times:

```
int i;
for (i = 0; i < 5; i++)
   System.out.println("Java is fun!");
```

This is equivalent to

```
int i = 0;
while (i < 5)
{
   System.out.println("Java is fun!");
   i++;
}
```

The initialization clause of a `for` loop can include both the declaration and initialization of the loop control variable:

```
for(int i = 0; i < 5; i++)
   System.out.println(“Java is fun!”);
```

This is a concise and handy way to declare simple loop control variables. This feature is not supported for the relational test clause of `for` or `while` loops. If the loop control variable `i` exists only for the duration of the `for` statement. We will elaborated on this later in our discussion of variable scope.

Sometimes the loop control variable is initialized "high" and then adjusted "backward". This is illustrated in the `StringBackwards` demo program in Figure 6.

```
 1  public class StringBackwards
 2  {
 3     public static void main(String[] args)
 4     {
 5        String s = "Hello, world";
 6        int length = s.length();
 7        for (int i = length; i > 0; i--)
 8           System.out.print(s.substring(i - 1, i));
 9     }
10  }
```

Figure 6. `StringBackwards.java`

The output of this program is

```
dlrow ,olleH
```

In this example, the loop control variable `i` is initialized with the length of the string (lines 6-7). As the loop progresses, `i` is decremented using the postfix decrement operator. The loop terminates when the relational expression "`i > 0`" is `false`. Recall from our discussion of strings in Chapter 2 that the length of a string is "one greater than" the index of the last character in a string. For this reason, the `substring()` method in line 8 retrieves the character starting at `i - 1`, rather than `i`.

Line 6 is not necessary in `StringBackwards.java`, but it was included to avoid clutter in the `for` loop. The loop could just as easily take the following form

```
for (int i = s.length(); i > 0; i--)
   System.out.print(s.substring(i - 1, i));
```

The adjustment expression needn't be limited to a simple increment or decrement of the loop control variable. The following `for` loop counts by threes from zero to one hundred:

```
for (int i = 0; i < 100; i += 3)
   System.out.println(i);
```

The adjustment expression "`i += 3`" increments the loop control variable by three after each print statement.

Loop control needn't be restricted to expressions involving the loop control variable. The following loop prints the cubes of integers starting at zero, increasing until the result approaches but does not exceed 500.

```
int j = 0;
for (int i = 0; j <= 500; i++)
{
   System.out.println(i + "\t" + j);
   j = i * i * i;
}
```

The output is

```
0    0
1    1
2    8
3    27
4    64
5    125
6    216
7    343
```

Since we want to terminate the loop "when the *result* approaches but does not exceed 500" the test does not use the loop control variable, `i`. The loop control relational expression "`j <= 500`" ensures the loop stops when the result – an integer cube – approaches but does not exceed 500. Note the use of the tab character escape sequence (`\t`) to conveniently align the two columns of output.

Comma Operator

It is possible to include more than one expression in the loop initialization and loop adjustment parts of the `for` statement. This is a accomplished using the comma (`,`) operator. The previous example of printing integer cubes could be coded as follows using the comma operator:

```
for (int i = 0, j = 0; j < 500; i++, j = i * i * i)
   System.out.println(i + "\t" + j);
```

Although use of the comma operator is never essential, it occasionally springs to mind as a convenient way to setup a simple loop.

break Statement

We have already met the `break` statement as a means to exit a `switch` after the statements for a selected "case" execute. The break statement also works as an "early" exit from a loop; that is, to exit *before* the loop control relational expression yields "`false`". Suppose we want to process characters in a string starting at the beginning of the string and proceeding through the string. However, assume we want to stop as soon as a period (`.`) is read. The program `DemoBreak` shows how this can be done (see Figure 7).

```
 1  import java.io.*;
 2
 3  public class DemoBreak
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        BufferedReader stdin =
 8           new BufferedReader(new InputStreamReader(System.in), 1);
 9
10        System.out.println("Enter a line of characters:");
11        String s = stdin.readLine();
12        for (int i = 0; i < s.length(); i++)
13        {
14           if (s.charAt(i) == '.')
15              break;
16           else
17              System.out.print(s.charAt(i));
18        }
19        System.out.println("\nThank you");
20     }
21  }
```

Figure 7. `DemoBreak.java`

A sample dialogue with this program follows:

```
PROMPT>java DemoBreak
Enter a line of characters:
Yes! No. Maybe?
Yes! No
Thank you
```

The `for` loop is setup in line 12 to process the line character-by-character. Within the loop (lines 12-18), the first task is to determine if the character at index `i` is a period (line 14). If so, the `break` statement in line 15 executes and the loop is immediately exited. In this demo, "processing" is the simple act of printing each character (line 17).

A `switch` statement cannot be setup properly without using `break` statements. However, as a means to exit a loop early, a `break` statement is never essential. A workaround is always possible. For example, lines 12-18 in `DemoBreak.java` can be replaced with

```
for (int i = 0; i < s.length() && !(s.charAt(i) == '.'); i++)
   System.out.print(s.substring(i, i + 1));
```

This approach saves a few lines of code, but it is also harder to understand. Note the use of lazy evaluation in the loop control relational expression to ensure that the `charAt()` method does not execute unless the index `i` represents a valid character in the string.

There is an on-going debate in the computer science community on whether the `break` statement is good news or bad news. As it turns out, programs that use `break` statements to exit loops are extremely difficult (arguably "impossible") to verify for correctness. In very large, safety-critical projects, it is probably best to avoid the `break` statement. However, in our ever-expanding bag of Java programming tools, the `break` statement is often the best "fit" for our mental model of a problem. We will continue to use `break` statements in these notes.

By definition, the `break` statement exits the inner-most loop or `switch` where it is located. If a `break` statement is located in a `switch` statement that is inside a loop, it serves only to exit the

`switch` statement. If a `break` statement is inside a loop that is inside another loop, it serves only to exit the inner loop.

continue Statement

The `continue` statement provides a mechanism to skip statements in a loop and proceed directly to the loop control relational expression. The program `DemoContinue` inputs a line of characters and then processes (i.e., prints) only the digit characters in the line (see Figure 8).

```
 1  import java.io.*;
 2
 3  public class DemoContinue
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        BufferedReader stdin =
 8           new BufferedReader(new InputStreamReader(System.in), 1);
 9
10        System.out.println("Enter a line of characters:");
11        String s = stdin.readLine();
12        for (int i = 0; i < s.length(); i++)
13        {
14           if (s.charAt(i) < '0' || s.charAt(i) > '9')
15              continue;
16           System.out.print(s.charAt(i));
17        }
18        System.out.println("\nThank you");
19     }
20  }
```

Figure 8. `DemoContinue.java`

A sample dialogue follows:

```
PROMPT>java DemoContinue
Enter a line of characters:
Process only 0123456789 characters
0123456789
Thank you
```

The character is processed (i.e., printed) character by character; however, line 14 determines if the current character is a digit. If it is not, the continue statement in line 15 executes to immediately begin processing the next character.

A `continue` statement can be avoided by reversing the relational test and adding another level of indentation to the statement(s) that follows. Lines 14-16 in `DemoContinue` can be replaced with

```
if ( !(s.charAt(i) < '0' || s.charAt(i) > '9') )
   System.out.print(s.charAt(i));
```

The `!` (NOT) logical operator reverses the relational test. Note that the print statement above is indented farther than the same statement in line 16 in Figure 8. The difference is minor; however, if the print statement is replaced by a large statement block, the cosmetic effect of the added indentation may be undesirable. The `continue` statement can help prevent nested loops, etc., from creeping too far to the right, with lines extending off the viewable area of the editor's display.

For the most part `continue` is used simply because it is the most convenient tool at hand to solve a particular programming problem. If it suits your purpose, by all means use it.

Note that a `continue` statement has no particular use in a `switch` statement (unlike `break`). If a `continue` statement does appear in a `switch` statement, execution proceeds to the loop control relational expression of the inner-most loop containing the `switch`.

Infinite Loops

Infinite loops were mentioned earlier as a possible side effect of incorrectly setting up a loop control relational expression. In these situations, the infinite loop is definitely a programming bug, because the program never terminates. However, infinite loops can be exploited as a simple way to set-up a loop that executes indefinitely until some condition occurs.

The program `DemoInfiniteLoop` reads lines from the standard input and echoes the lines to the standard output. This continues in a loop until a line beginning with 'Q' is inputted (see Figure 9).

```
 1  import java.io.*;
 2
 3  public class DemoInfiniteLoop
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        BufferedReader stdin =
 8           new BufferedReader(new InputStreamReader(System.in), 1);
 9
10        System.out.println("Enter lines of characters:\n" +
11                           "(Line beginning with 'Q' quits)");
12        while (true)
13        {
14           String s = stdin.readLine();
15           if (s.length() > 0 && s.charAt(0) == 'Q')
16              break;
17           System.out.println(s);
18        }
19        System.out.println("Thank you");
20     }
21  }
```

Figure 9. `DemoInfiniteLoop.java`

A sample dialogue with this program follows:

```
PROMPT>java DemoInfiniteLoop
Enter lines of characters:
(Line beginning with 'Q' quits)
Hello
Hello
Good bye
Good bye
Quit
Thank you
```

This program uses a `while` loop of the following form:

```
while (true)
   statement;
```

Since the boolean constant "`true`" is always "`true`", the loop is an infinite loop. The trick is to use a `break` statement to exit the loop when a certain condition occurs. In this example, that condition is a line beginning with 'Q'. Note the use of lazy evaluation in line 15 to ensure that the `charAt()` method doesn't execute if a blank line is inputted.

Setting up infinite loops, as above, is simple and effective. As with `break`, `continue`, and the comma operator, there is always an alternate way to create the same effect. Of course, if you are of the conviction that `break` statements are bad news, then you are unlikely to buy-into the deliberate use of infinite loops, as above.

For completeness, let's examine two alternatives to the `while` loop in `DemoInfiniteLoop`. Lines 12-18 can be replaced with the following statements:

```
boolean moreData = true;
while (moreData)
{
   String s = stdin.readLine();
   if (s.length() > 0 && s.charAt(0) == 'Q')
      moreData = false;
   else
      System.out.println(s);
}
```

In this approach a `boolean` variable `moreData` is used as a "flag" to signal the end of input. The initial state of `moreData` is `true`, so the loop executes at least once. When a line is inputted that begins with 'Q' `moreData` is set to `false`. The next time the loop control relational expression is evaluated, the result is `false` and the loop terminates.

For the ultimate in dense code, the same lines can be further reduced as follows:

```
String s;
while ((s = stdin.readLine()).length() != 0 && !(s.charAt(0) == 'Q'))
   System.out.println(s);
```

The `while` loop now includes a large loop control relational expression containing an assignment expression, `(s = stdin.readLine())`. This is perfectly legal; however, since assignment has the lowest precedence, parentheses are added. The result is simply the string assigned and, so, it is also perfectly legal to append "`.length()`" to determine the length of the string. Further appending "`!= 0`" simply implements a relational expression which is `true` as long as the line inputted is not a blank line. This is followed by the logical && operator. Lazy evaluation ensures that the right-hand argument is not evaluated unless a non-blank line was inputted. If indeed the line was non-blank, the expression `!(s.charAt(0) == 'Q')` further ensures that the line does not begin with 'Q'. If the final result of this mega-expression is `true`, the inputted line is printed on the standard output in the next line — and then we start all over again!

We have just examined three ways to solve the same problem. The first used an infinite loop with a break statement. The second used a `boolean` flag to exit a loop. The third used a one-statement loop with a very dense loop control relational expression. So, which of these is "the best". If you are using these notes as a student in a Java programming course, your instructor will, no doubt, have an opinion on this. Your best bet is to adopt the style suggested by your

instructor. In fact, subscribing to, and sticking with, a consistent programming philosophy is the best advice we can offer.

Variable Scope

Earlier, we noted that a variable can be declared anywhere, provided it is declared before it is used. This seems simple enough, and it is, as long as programs execute in a straight line. Now, our programs include choices and loops. Due to a characteristic known as *variable scope*, a bit more attention is needed on the placement of variable declarations when loops are involved. Once a variable is defined it exists from that point on — *but only within the block where it is defined*. This point is illustrated using a simple demo program (see Figure 10).

```
 1  public class DemoVariableScope
 2  {
 3     public static void main(String[] args)
 4     {
 5        String advice = "Haste makes waste";
 6        int i;
 7        for (i = 0; i < 2; i++)
 8           System.out.println(advice);
 9        System.out.println("Good advice is worth giving...");
10        System.out.println(i + " times");
11     }
12  }
```

Figure 10. `DemoVariableScope.java`

This program generates the following output:

```
Haste makes waste
Haste makes waste
Good advice is worth giving...
2 times
```

The ”2" in the final line of output is the value of the `int` variable `i`. Now, have a look at line 6 in the listing. Note that the `int` variable `i` was declared *before* the `for` loop. This is important. If `i` were defined inside the initialization of the `for` loop, as commonly done, a compile error will occur at line 10, because `i` only exists for the duration of the loop. The error message will indicate "`Undefined variable: i`". This is illustrated in Figure 11.

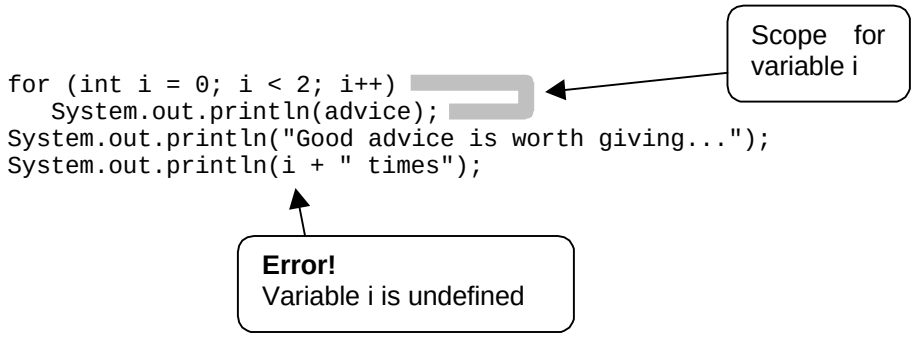


Figure 11. Variable scope example

If a variable is not needed after the loop, then the approach in Figure 11 is fine.

For nested loops, even more care is warranted. The same rule applies, but greater opportunity for error exists. An example of variable scope with nested loops is shown in Figure 12.

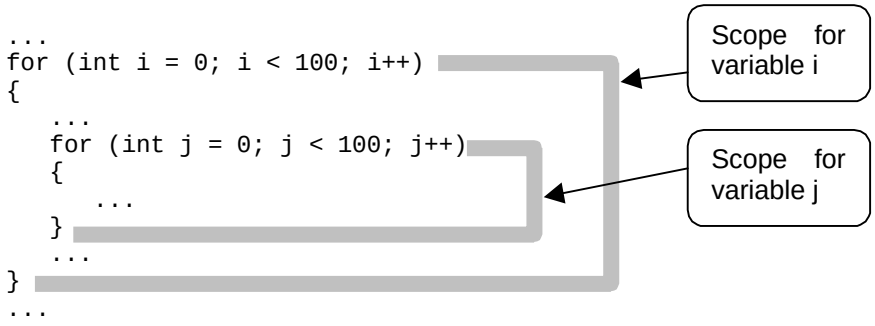


Figure 12. Variable scope for nested loops

This completes our discussion of "choice" and "loops" in the Java language. In summary,

- "Choices" may be implemented using the `if` statement or the `switch` statement.
- "Loops" may be implemented using the `while`, `do/while`, or `for` statements.
- A `break` statement is used to immediately exit a `switch` or a loop.
- A `continue` statement inside a loop causes an immediate branch to the loop control relational expression.
- A variable has scope, or visibility, only in the block where it is defined.

Chapter 4

Working With Java

Organization of Java

With a firm grounding in the elements of the Java language and in designing programs that include choices and loops, we are well equipped to put Java to work in solving interesting problems. In this section, we will move further along in our study of Java, as we learn to use additional classes in Java's Application Programming Interface (API). At this stage, we only want to put classes to work in interesting programs. Many advanced topics, such as inheritance and polymorphism, are deferred to later chapters.

The Java API is a collection of *packages*. Each package contains a collection of *classes*, and each class contains a collection of *methods*. Through methods, *objects* are created and acted upon. In the following sections, we will describe the organization of Java, as seen through its packages, classes, methods, and objects.

What is a Package?

A *package* is a collection of classes. Packages are simply a convenient place for developers to put classes of a common purpose; however, they play no specific role in the Java language. The Java developers at Sun Microsystems have organized the many hundreds of classes in the Java API into just over 50 core packages. Within these, hundreds of classes and thousands of methods are defined. We can define our own classes and put them into a package as well, and we'll learn how to do that later.

The packages from the Java API encountered in these notes are summarized in Table 1.

Table 1. Packages in the Java API

Package	Description
`java.applet`	provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context
`java.awt`	contains all of the classes for creating user interfaces and for painting graphics and images ("awt" is an acronym for “advanced windowing toolkit”)
`java.io`	provides for system input and output through data streams, serialization, and the file system
`java.lang`	provides classes that are fundamental to the design of the Java programming language
`java.util`	contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array)
`javax.swing`	provides a set of "lightweight" (all-Java language) components that, to the maximum degree possible, work the same on all platforms

The import Statement

To use classes from a package in the Java API, the `import` statement is required at the top of the program. For example, the `BufferedReader` class appears in the `java.io` package. Any program that uses this class must include the following statement at the top of the source file:

```
import java.io.BufferedReader;
```

The `import` statement informs the compiler of the whereabouts of classes used in a program that are not defined in the program. It is common and convenient to simplify the statement above as follows:

```
import java.io.*;
```

The asterisk (`*`) is a wildcard character. With this statement, any class from the `java.io` package can be used in the program.

The `java.lang` package is special. It provides classes that are fundamental to the design of the Java language. The most important classes are `Object`, which is the root of the class hierarchy, and `Class`, instances of which represent classes at run time. In most cases, we are not aware of the presence or use of these classes, as they operate "behind the scenes" in a program. As well, the `java.lang` package provides the wrapper classes, the `String` class, and the `System` class. We will say more about these later.

Since the `java.lang` package is so fundamental to the language, the following statement is implied for all Java programs:

```
import java.lang.*;  // Implied!  Not necessary
```

What is a Class?

A class is like a data type. Just as we create and operate on variables of a certain data type, we create and operate on objects of a certain class. However, a class is not a primitive data type, like `int` or `double`, because the data it encapsulates are more sophisticated. Central to this sophistication are the methods of each class. So, for example, there is a class named "`String`", and we can use methods of the `String` class to perform useful operations on `String` objects.

However, not all classes fit the description above. Each of our demo programs contained a class bearing the same name as the filename. Clearly, these classes are in no way a "data type". As well, Java's `Math` class is unlike a data type. One cannot create a `Math` object (like we create a `String` object). The `Math` class is simply a convenient holding place for useful methods (to perform mathematical operations) and useful data fields (like the constant `PI`).

One feature that is common to all classes is that they contain methods. From the above discussion, it appears there are two broad categories of classes: those for which objects can be created, and those for which objects cannot be created. The `String` class is an example of the former, the `Math` class, the latter. This is a useful distinction, and we will draw on it throughout this discussion.

It's hard to talk about classes without mentioning objects and methods. Let's examine each of these.

What is an Object?

An object is an instance of a class. The act of "instantiating an object" is the act of creating an object of a class and initializing its data fields with values. So, a `String` object is an instance of the `String` class. Once an object is instantiated, we can perform operations on it. However, unlike a variable representing a primitive data type, we cannot, in general, directly access the data fields of an object. Our only means to do so is through the methods of that object's class.

What is a Method?

A method is a set of instructions to do something. So, in a programming sense, methods do the work, and objects are the things methods work on. We can categorize all Java methods as being *constructor methods*, *instance methods*, or *class methods*. Let's look at each of these.

Constructor Methods

The purpose of a constructor method is to construct an object. This is often expressed in more formal terms: The purpose of a constructor method is to *instantiate an object*. It seems reasonable from our earlier discussion that the `Math` class does not have a constructor method (One cannot instantiate a `Math` object!), whereas the `String` class does. In fact, this is the case. There is no method called `Math()`. However, there is a method called `String()`. It is a rule of the Java language that a constructor method has the same name as its class. We have used a few constructor methods already, such as `String()`, `BufferedReader()`, and `InputStreamReader()`. These are constructor methods for the classes `String`, `BufferedReader`, and `InputStreamReader`, respectively.

A characteristic of Java and other object-oriented programming languages is the ability to have more than one method with the same name. We saw examples earlier of two methods called `String()`. One was called without an argument inside the parentheses, the other was called with one argument — a string constant. These are two different methods. The compiler determines which to use by context. If the compiler finds a `String()` constructor with no argument, the default constructor is used. If one argument appears and it is a string constant, then the method with one string constant is used. When two methods have the same name, the methods are said to be *overloaded*. A more fancy term for this is *polymorphism*, which is discussed later.

Instance Methods and Class Methods

Distinguishing between instance methods and class methods is a bit tricky. Most of the methods used in demo programs thus far were instance methods. Consider the following two statements:

```
String s = new String("hello");
int len = s.length();
```

There are two methods above: a constructor method named `String()` and an instance method named `length()`. Both are methods of the `String` class. The `length()` method is an instance method because it is called through an "instance" of the `String` class (in this case, `s`). Dot notation connects the implicit variable's name to the method's name. So, if an object variable precedes the method's name, the method is an instance method: It is called through an instance of the class.

Now consider the following two statements:

```
double x = 99.0;
double y = Math.sqrt(x);
```

The method `sqrt()` is a class method. (Class methods are also called *static methods*.) Preceding the method's name is "`Math.`" which is the name of a class. So, what precedes the method's name helps distinguish a class method from an instance method.

In some cases, class methods are called without dot notation. In fact, the only reason the prefix "`Math.`" is required, is to identify to the compiler where the `sqrt()` method is defined. (It is

defined in the `Math` class in the `java.lang` package.) If a method is defined in the same file where it is called, then prefixing the method's name with the class name is unnecessary. We will see examples of this later.

Class Hierarchy

Although we'll say a great deal about class hierarchies later, this important topic deserves a quick tour now to alert you to an essential characteristic of Java and other object-oriented programming languages. We noted earlier that the relationship between a class and its package is mostly one of convenience. (A package is a convenient place to put common classes.) Of far more importance is the relationship of a class to other classes. All classes in Java exist in a hierarchy. Those "further down" the hierarchy inherit characteristics from those "further up" the hierarchy.

At the top of the hierarchy is the `Object` class. All Java classes inherit characteristics from the `Object` class. An example of a class "just below" the `Object` class is the `String` class. In this relationship, the `Object` class in the *superclass*, and the `String` class is a *subclass*. Figure 1a illustrates this. By convention, the arrow points to the superclass. It is common to show class hierarchies horizontally, as in Figure 1b, since more classes can be placed in a single illustration. We'll use this format in these notes.

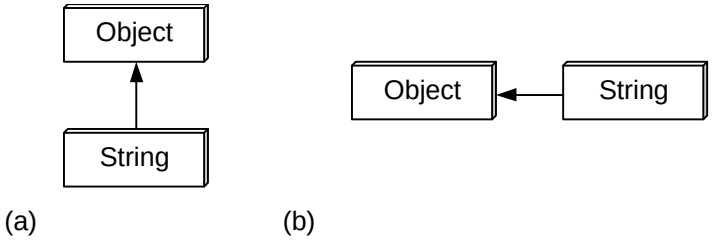


Figure 1. Class hierarchy (a) vertical illustration (b) horizontal illustration

The documentation provided by Sun Microsystems for the Java API illustrates this relationship as shown in Figure 2.

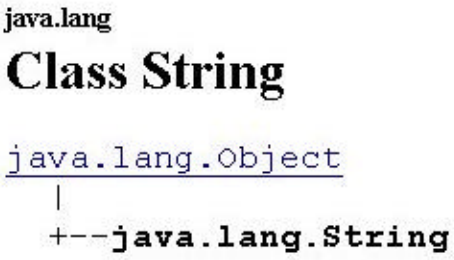


Figure 2. Class hierarchy, as shown in the Java API documentation

Since the `String` class is a subclass of the `Object` class, `String` objects inherit characteristics of `Object` objects. An important benefit of this relationship is that methods defined in the `Object` class are automatically methods of the `String` class too. The `String` class inherits these as part and parcel of the inheritance relationship. We'll say more about this later.

With this introduction to classes, objects, methods, packages, and class hierarchy, we are well prepared to explore some of the many interesting classes in the Java language.

Wrapper Classes

Each of Java's eight primitive data types has a class dedicated to it. These are known as *wrapper classes*, because they "wrap" the primitive data type into an object of that class. So, there is an `Integer` class that holds an `int` variable, there is a `Double` class that holds a `double` variable, and so on. The wrapper classes are part of the `java.lang` package, which is imported by default into all Java programs. The following discussion focuses on the `Integer` wrapper class, but applies in a general sense to all eight wrapper classes. Consult the Java API documentation for more details.

The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

```
int     x = 25;
Integer y = new Integer(33);
```

The first statement declares an `int` variable named `x` and initializes it with the value 25. The second statement instantiates an `Integer` object. The object is initialized with the value 33 and a reference to the object is assigned to the object variable `y`. The memory assignments from these two statements are visualized in Figure 1.

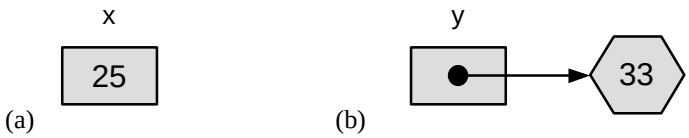


Figure 1. Variables vs. objects (a) declaration and initialization of an `int` variable (b) instantiation of an `Integer` object

Clearly `x` and `y` differ by more than their values: `x` is a variable that holds a value; `y` is an object variable that holds a reference to an object. As noted earlier, data fields in objects are not, in general, directly accessible. So, the following statement using `x` and `y` as declared above is not allowed:

```
int z = x + y;  // wrong!
```

The data field in an `Integer` object is only accessible using the methods of the `Integer` class. One such method — the `intValue()` method — returns an `int` equal to the value of the object, effectively "unwrapping" the `Integer` object:

```
int z = x + y.intValue();  // OK!
```

Note the format of the method call. The `intValue()` method is an instance method, because it is called through an instance of the class — an `Integer` object. The syntax uses dot notation, where the name of the object variable precedes the dot.

Some of Java's classes include only instance methods, some include only class methods, some include both. The `Integer` class includes both instance methods and class methods. The class methods provide useful services; however, they are not called through instances of the class. But, this is not new. We met a class method of the `Integer` class earlier. The statement

```
int x = Integer.parseInt("1234");
```

converts the string "`1234`" into the integer 1,234 and assigns the result to the `int` variable `x`. The method `parseInt()` is a class method. It is *not* called through an instance of the class.

Although dot notation is used above, the term "`Integer.`" identifies the class where the method is defined, rather than name of an `Integer` object. This is one of the trickier aspects of distinguishing instance methods from class methods. For an instance method, the dot is preceded by the name of an object variable. For a class method, the dot is preceded by the name of a class.

It is interesting that the `parseInt()` method accepts a `String` argument and returns an `int`. Neither the argument nor the returned value is an `Integer` object. So, the earlier comment that class methods "provide useful services" is quite true. They do not operate on instance variables; they provide services that are of general use. The `parseInt()` method converts a `String` to an `int`, and this is a useful service — one we have called upon in previous programs.

Another useful `Integer` class method is the `toString()` method. The following statement

```
String s = Integer.toString(1234);
```

converts the `int` constant 1,234 to a `String` object and returns a reference to the object. We have already met this method through the following shorthand notation:

```
String s = "" + 1234;
```

In general, the data fields in objects are private and are only accessible through methods of the class. However, class definitions sometimes include constants that are "public", and, therefore, accessible anywhere the class is accessible. Java's wrapper classes include publicly defined constants identifying the range for each type. For example, the `Integer` class has constants called `MIN_VALUE` and `MAX_VALUE` equal to 0x80000000 and 0x7fffffff respectively. These values are the hexadecimal equivalent of the most-negative and most-positive values represented by 32-bit integers. The table of ranges shown earlier was obtained from a simple program called `FindRanges` that prints these constants. The listing is given here in Figure 2.

```
 1  public class FindRanges
 2  {
 3     public static void main(String[] args)
 4     {
 5        System.out.println("Integer range:");
 6        System.out.println("   min: " + Integer.MIN_VALUE);
 7        System.out.println("   max: " + Integer.MAX_VALUE);
 8        System.out.println("Double range:");
 9        System.out.println("   min: " + Double.MIN_VALUE);
10        System.out.println("   max: " + Double.MAX_VALUE);
11        System.out.println("Long range:");
12        System.out.println("   min: " + Long.MIN_VALUE);
13        System.out.println("   max: " + Long.MAX_VALUE);
14        System.out.println("Short range:");
15        System.out.println("   min: " + Short.MIN_VALUE);
16        System.out.println("   max: " + Short.MAX_VALUE);
17        System.out.println("Byte range:");
18        System.out.println("   min: " + Byte.MIN_VALUE);
19        System.out.println("   max: " + Byte.MAX_VALUE);
20        System.out.println("Float range:");
21        System.out.println("   min: " + Float.MIN_VALUE);
22        System.out.println("   max: " + Float.MAX_VALUE);
23     }
24  }
```

Figure 2. `FindRanges.java`

This program generates the following output:

```
Integer range:
   min: -2147483648
   max: 2147483647
Double range:
   min: 4.9E-324
   max: 1.7976931348623157E308
Long range:
   min: -9223372036854775808
   max: 9223372036854775807
Short range:
   min: -32768
   max: 32767
Byte range:
   min: -128
   max: 127
Float range:
   min: 1.4E-45
   max: 3.4028235E38
```

In the case of `float` and `double`, the minimum and maximum values are those closest to plus or minus zero or plus or minus infinity, respectively.

The most common methods of the `Integer` wrapper class are summarized in Table 1. Similar methods for the other wrapper classes are found in the Java API documentation.

Table 1. Methods of the Integer Wrapper Class

Method	Purpose
Constructors	
`Integer(i)`	constructs an `Integer` object equivalent to the integer `i`
`Integer(s)`	constructs an `Integer` object equivalent to the string `s`
Class Methods	
`parseInt(s)`	returns a signed decimal integer value equivalent to string `s`
`toString(i)`	returns a new `String` object representing the integer `i`
Instance Methods	
`byteValue()`	returns the value of this `Integer` as a `byte`
`doubleValue()`	returns the value of this `Integer` as an `double`
`floatValue()`	returns the value of this `Integer` as a `float`
`intValue()`	returns the value of this `Integer` as an `int`
`longValue()`	returns the value of this `Integer` as a `long`
`shortValue()`	returns the value of this `Integer` as a `short`
`toString()`	returns a `String` object representing the value of this `Integer`

In the descriptions of the instance methods, the phrase "this `Integer`" refers to the instance variable on which the method is acting. For example, if `y` is an `Integer` object variable, the expression `y.doubleValue()` returns the value of "this `Integer`", or `y`, as a `double`.

The Math Class

It is hard to avoid the `Math` class in any Java program that requires scientific or other numeric computations. As with the wrapper classes, the `Math` class is part of the `java.lang` package; so, methods may be used without an explicit `import` statement. Table 1 summarizes the most common methods in the `Math` class. For a full listing, see the Java API documentation. Unlike the wrapper classes, the `Math` class contains no constructor method: `Math` objects cannot be instantiated. Therefore, all methods in Table 1 are class methods (aka static methods).

Table 1. Methods in the `Math` Class

Method	Purpose
`abs(double a)`	returns a `double` equal to the absolute value of a `double` value `a`
`asin(double a)`	returns a `double` equal to the arc sine of an angle `a`, in the range of -pi/2 through pi/2
`exp(double a)`	returns a `double` equal to the exponential number e (i.e., 2.718...) raised to the power of a `double` value `a`
`log(double a)`	returns a `double` equal to the natural logarithm (base e) of a `double` value `a`
`pow(double a, double b)`	returns a `double` equal to of value of the first argument `a` raised to the power of the second argument `b`
`random()`	returns a `double` equal to a random number greater than or equal to 0.0 and less than 1.0
`rint(double a)`	returns a `double` equal to the closest `long` to the argument `a`
`round(double a)`	returns a `long` equal to the closest `int` to the argument `a`
`sin(double a)`	returns a `double` equal to the trigonometric sine of an angle `a`
`sqrt(double a)`	returns a `double` equal to the square root of a `double` value `a`
`tan(double a)`	returns a `double` equal to the trigonometric tangent of an angle `a`
`toDegrees(double a)`	returns a `double` equal to an angle measured in degrees equal to the equivalent angle measured in radians `a`
`toRadians(double a)`	returns a `double` equal to an angle measured in radians equal to the equivalent angle measured in degrees `a`

As well as methods, class definitions often include publicly-accessible variables or constants as "fields". The `Math` class includes two such fields: `E` and `PI`, representing common mathematical constants (see Table 2).

Table 2. Data Fields in the `Math` Class

Constant	Type	Purpose
`E`	`double`	holds the value of e, the base of natural logarithms,

		accurate to about 15 digits of precision
`PI`	`double`	holds the value of pi, the ratio of the circumference of a circle to its radius, accurate to about 15 digits or precision

To use these constants in an expression, they must be prefixed with "`Math.`" (e.g., `Math.PI`) to inform the compiler of the whereabouts of their declarations. Note the use of uppercase characters for the identifiers. This is consistent with the naming conventions for constants, as given earlier.

Let's demonstrate a few of the methods in the `Math` class through demo programs.

Example: Equal-Tempered Musical Scale

The harmonic system used in most Western music is based on the "equal-tempered scale", as typified by the pattern of white and black keys on a piano. Each note on a piano keyboard is related in frequency (f) to its neighbor by the following formula:

$$f_{n+1} = f_n \times 2^{1/12}$$

where note $n + 1$ is "one semitone" above note n. The factor $2^{1/12}$ ensures that two notes separated by twelve steps in the equal-tempered scale differ in frequency by a factor of $2^{12/12} = 2$. This interval is known as an "octave". To allow musicians to travel and perform with different orchestras in different countries, a standard evolved and was adopted in the 1950s to ensure instruments were in tune with each other. The standard specifies that "A above middle C" should have a frequency of 440 Hz (see Figure 1). With this reference point, and with the $2^{1/12}$ frequency factor between adjacent notes, the frequency of any note on any instrument was standardized.

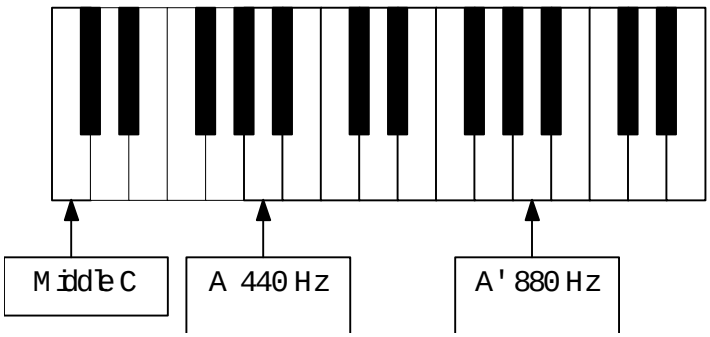


Figure 1. Notes and frequencies in the equal-tempered scale

As seen in Figure 1, the frequency of A' — one octave above A — is $440 \times 2^{12/12} = 880$ Hz. But what about the notes in between? Their frequencies are computed in the demo program `Frequencies` (see Figure 2).

```
 1  public class Frequencies
 2  {
 3     public static void main(String[] args)
 4     {
 5        for (int i = 0; i < NOTES.length; ++i)
 6        {
 7           System.out.print(NOTES[i] + "  ");
 8           System.out.println(A440 * Math.pow(2, i / 12.0));
 9        }
10
11     }
12     public static final double A440 = 440.0;
13     public static final String[] NOTES =
14        {  "A ", "A#", "B ", "C ", "C#", "D ",
15           "D#", "E ", "F ", "F#", "G ", "G#", "A'" };
16  }
```

Figure 2. `Frequencies.java`

This program generates the following output:

```
A   440.0
A#  466.1637615180899
B   493.8833012561241
C   523.2511306011972
C#  554.3652619537442
D   587.3295358348151
D#  622.2539674441618
E   659.2551138257398
F   698.4564628660078
F#  739.9888454232688
G   783.9908719634985
G#  830.6093951598903
A'  880.0
```

The frequencies in the table above are computed iteratively for 13 notes beginning at 440 Hz (see line 8).[1] The calculation uses the `pow()` method in the `Math` class. Two arguments are required. The first is the base and the second is the power. The base is 2 in each case, but the power is increased each time through the loop to compute the frequency of the next note. Note the use of "`12.0`" in line 8. The "`.0`" is important, as it ensures the `int` variable `i` is promoted to a `double`.

In musical terms, the most consonant harmony is formed by playing two notes where the higher note is 1.5 times the frequency of the lower note. This interval is known as a musical "fifth" and it is equivalent to seven semitones in the equal-tempered scale. A fifth above A is E. Note in the table above that the frequency of E is 659.255 Hz. This is close to $1.5 \times 440 = 660$ Hz. The slight discrepancy is not due to a rounding error. In fact, the equal-tempered scale is a compromise over a system where notes are related by natural harmonic intervals. The difference between 659.225 Hz and 660 Hz is small, but important. One reason pianos are so difficult to tune is that each note must conform to the equal-tempered scale, so slight discrepancies from natural harmonic intervals must be tuned-in.

[1] The names of the notes are stored as an array of strings. Arrays are discussed in Chapter 6.

More information on the equal-tempered scale and other systems of tuning, see Backus' *The Acoustical Foundations of Music* [2].

Example: Calculating the Height of a Building

Our next example of methods in the `Math` class shows how to a surveyor might calculate the height of a building or other object given two measurements. One measurement is the surveyor's distance from the building, the other is the line-of-sight angle between the ground (which we assume is flat) and the roof of the building (see Figure 3).

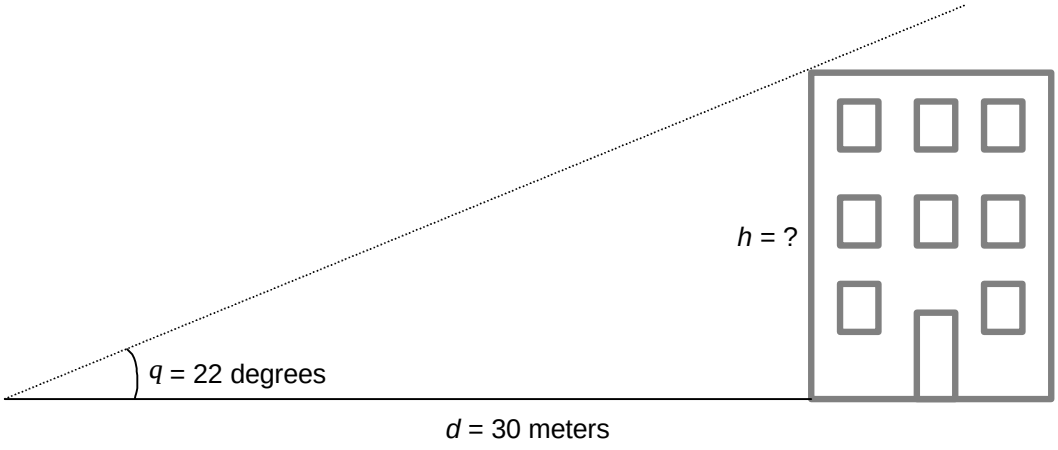


Figure 3. Calculating the height of a building

Figure 4 reviews the geometry. Clearly, x is the height of the building, y is the surveyor's distance from the building, and q is the line-of-sight angle between the ground and the roof. Our task is to compute $x = y \tan(q)$.

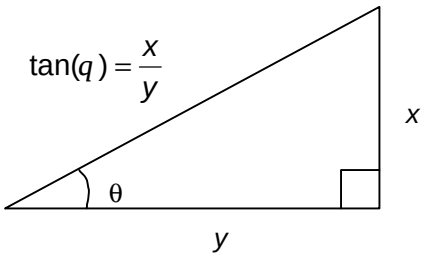


Figure 4. Review of tangent function

The demo program `Height` prompts the user to enter a distance in meters and a line-of-sight angle in degrees. From these, the height of the building is calculated.

```
 1  import java.io.*;
 2
 3  public class Height
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        BufferedReader stdin =
 8           new BufferedReader(new InputStreamReader(System.in), 1);
 9
10        // Input distance to building
11        System.out.print("Enter distance to building: ");
12        double distance = Double.parseDouble(stdin.readLine());
13
14        // Input line-of-sight angle to top of building
15        System.out.print("Enter line-of-sight angle (degrees) to roof: ");
16        double angle = Double.parseDouble(stdin.readLine());
17
18        // Calculate height of building
19        double aRadians = Math.toRadians(angle);
20        double height = distance * Math.tan(aRadians);
21
22        // print height
23        System.out.println("Building is " + height + " meters high");
24     }
25  }
```

Figure 5. `Height.java`

A sample dialogue with this program follows: (User input is underlined.)

```
PROMPT>java Height
Enter distance (meters) to building: 30
Enter line-of-sight angle (degrees) to roof: 22
Building is 12.120786775054704 meters high
```

Two methods in the `Math` class are used. The `tan()` method computes the tangent of an angle passed to it as a `double` argument (line 20). However, the `tan()` method requires an angle in radians as an argument. The `toRadians()` method converts the inputted angle from degrees to radians (see line 19).

Example: Charging a Capacitor

In electronics, a well-known phenomenon is the charging characteristic of a capacitor. A representative circuit is shown in Figure 6. Capacitor *C* is in series with resistor *R* and a voltage V_i is applied to the *RC* pair through a switch.

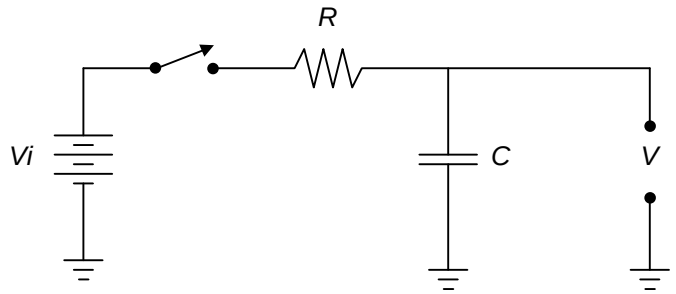


Figure 6. Charging a capacitor

Assuming an initial discharged state, the voltage across the capacitor, *V*, rises in time after the switch is closed according to the following formula:

$$V = V_i(1 - e^{-t/RC})$$

Let's examine a program to calculate *V* at several points in time after the switch is closed. To do this, we'll input sample values for *C*, *R*, and V_i.

The trickiest part in the calculation is raising the natural logarithm constant e to the power *-t / RC*. Java's `Math` class includes a data field, `Math.E`, that defines e to about 15 digits of precision, so we could use the `pow()` method, as we did in the `Frequencies` demo program (see Figure 2). However, the `exp()` method is a better choice. It computes e^n, where *n* is a `double` argument passed to the method. The demo program `CapicitorCharging` shows the `exp()` method in action.

```
 1  import java.io.*;
 2
 3  public class CapacitorCharging
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        BufferedReader stdin =
 8           new BufferedReader(new InputStreamReader(System.in), 1);
 9
10        // Input value of capacitor (micro Farads)
11        System.out.print("Capacitance (uF): ");
12        double c = Double.parseDouble(stdin.readLine()) * 1000000;
13
14        // Input value of resistor (in kilo Ohms)
15        System.out.print("Resistance (k): ");
16        double r = Double.parseDouble(stdin.readLine()) / 1000;
17
18        // Input value of input voltage (v)
19        System.out.print("Applied voltage (v): ");
20        double vi = Double.parseDouble(stdin.readLine());
21
22        System.out.println("\nt (us)\tvoltage (v)");
23        System.out.println("----------------------------");
24        for (int t = 1; t <= 1000000; t *= 10)
25        {
26           double v = vi * (1 - Math.exp( -t / (r * c)));
27           System.out.println(t + "\t" + v);
28        }
29     }
30  }
```

Figure 7. `CapacitorCharging.java`

A sample dialogue with this program follows:

```
PROMPT>java CapacitorCharging
Capacitance (uF): 1.5
Resistance (k): 33
Applied voltage (v): 12

t (us)      voltage (v)
---------------------------------
1            2.4242179371114503E-4
10           0.0024239975677593506
100          0.024217953426646677
1000         0.23999191951895327
10000        2.1950589463112236
100000      10.408455975563244
1000000     11.999999979790509
```

In the sample dialogue, the user specified a 1.5 μF capacitor in series with a 33 kΩ resistor with an applied voltage of 12 V. The output shows the capacitor voltage after 1 μs, 10 μs, 100 μs, and so on, up to 1,000,000 μs, or 1 second. The values inputted are in the most common units for capacitors (μF, or 10^{-6} farads) and resistors (kΩ or 10^{3} ohms). Within the program, these are converted to farads (line 12) and ohms (line 16).

For more examples of electronics circuits, see *The Art of Electronics* by Horowitz and Hill [3].

The StringTokenizer Class

In our demo programs thus far, input from the keyboard was processed one line at a time. Each line was converted to a `String` and operated on using available methods. However, it is often necessary to decompose a full line of input into *tokens* that are processed separately. Each token might be a word, for example. The `StringTokenizer` class in the `java.util` package provides services for this purpose.

By default, tokens are delimited by *whitespace* characters, but other delimiters are also possible. A whitespace is a space, return, tab, or newline character. The default delimiters are held in a string, `" \r\t\n"`, which can be changed using methods of the `StringTokenizer` class.

Unlike the `Math` class, objects of the `StringTokenizer` class can be instantiated; thus, the `StringTokenizer` class includes constructor methods. The methods of the `StringTokenizer` class are summarized in Table 1.

Table 1. Methods of the `StringTokenizer` Class

Method	Description
Constructors	
`StringTokenizer(String str)`	Constructs a string tokenizer for the specified string `str`; returns a reference the new object
`StringTokenizer(String str, String delim)`	Constructs a string tokenizer for the specified string `str` using `delim` as the delimiter set
`StringTokenizer(String str, String delim, boolean returnTokens)`	Constructs a string tokenizer for the specified string `str` using `delim` as the delimiter set; returns the tokens with the string if `returnTokens` is true
Instance Methods	
`countTokens()`	returns an `int` equal to the number of times that this tokenizer's `nextToken()` method can be called before it generates an exception
`hasMoreTokens()`	returns a `boolean` value: `true` if there are more tokens available from this tokenizer's string, `false` otherwise
`nextToken()`	returns a `String` equal to the next token from this string tokenizer
`nextToken(String delim)`	returns a `String` equal to the next token from this string tokenizer using `delim` as a new delimiter set

Count Words

Let's begin with a demo program. A basic tokenizing operation is to divide lines of input into words. The program `CountWords` demonstrates how this is done using methods of the `StringTokenizer` class (see Figure 1).

```
 1  import java.io.*;
 2  import java.util.*;
 3
 4  public class CountWords
 5  {
 6     public static void main(String[] args) throws IOException
 7     {
 8        BufferedReader stdin =
 9           new BufferedReader(new InputStreamReader(System.in), 1);
10
11        String line;
12        StringTokenizer words;
13        int count = 0;
14
15        // process lines until no more input
16        while ((line = stdin.readLine()) != null)
17        {
18           // process words in line
19           words = new StringTokenizer(line);
20           while (words.hasMoreTokens())
21           {
22              words.nextToken();
23              count++;
24           }
25        }
26        System.out.println("\n" + count + " words read");
27     }
28  }
```

Figure 1. `CountWords.java`

A sample dialogue with this program follows:

```
PROMPT>java CountWords
To be or not to be.
That is the question.
^z
10 words read
```

In the sample dialogue, two lines of text containing 10 words were inputted via the keyboard. The third line of input shows "`^z`". This means press and hold the Control key, then press `z`. When executing a Java program in a DOS window, `^z` indicates an end-of-input or end-of-file condition. When `^z` is detected, the `readLine()` method returns `null`. This condition signals that there is no more input available for processing. On systems running a *unix*-like operating system (e.g., *linux* or *Solaris*) an end-of-file condition is entered as `^d`.

In line 2, an `import` statement informs the compiler of the location of the `StringTokenizer` class. (It is in the `java.util` package.) Line 12 contains a declaration of an object variable named `words` of the `StringTokenizer` class. The `while` loop in lines 16-25 performs the task of inputting lines from the keyboard until a `null` (end-of-input) condition occurs. The relational expression in line 16 conveniently combines inputting a line of input with the relational test for `null`. Since assignment is of lower precedence than the "not equal to" (`!=`) relational operator, parentheses are required for the assignment expression.

Each line inputted is converted to a `String` object. A reference to the object is assigned to object variable `line` (lines 16). Line 19 instantiates a `StringTokenizer` object from the string `line` and assigns a reference to it to the object variable `words`. This sets-up the

conditions to begin processing the tokens — the words — in each line of text input. Two instance methods of the `StringTokenizer` class do all the work. The `hasMoreTokens()` method returns a `boolean` (`true` or `false`) indicating if there are more tokens to be processed. This method is called in line 20 via the instance variable `words` within the relational expression of an inner `while` loop. So, the outer `while` loop processes lines, the inner `while` loop processes the tokens in each line.

In the inner `while` loop, line 22 contains an expression to retrieve the next token in the `StringTokenizer` object `words`. Note that the token is retrieved, but is not assigned to anything. This is fine, because our task is simply to count words; we aren't interested in processing the words in any manner. The counting occurs in line 23, where the `int` variable `count` is incremented for each token retrieved. After both loops are finished the result is printed (line 26).[1]

Count Alpha-only Words

For out next example, we'll make a small but interesting change to `CountWords`. Suppose we want to count words if they contain only letters of the alphabet. To count only "alpha" words, we must not only tokenize strings into words, but each token must be inspected character-by-character to determine if all characters are letters. The demo program `CountWords2` shows the necessary modifications to perform this task (see Figure 2).

[1] The presence of a leading newline character ("\n") in the print statement is a fix for a quirk in *Windows*. If input arrives from the keyboard followed by Control-Z, the first line of subsequent output is overwritten by the operating system. The leading newline character ensures that the result printed is not overwritten.

```
 1  import java.io.*;
 2  import java.util.*;
 3
 4  public class CountWords2
 5  {
 6     public static void main(String[] args) throws IOException
 7     {
 8        BufferedReader stdin =
 9           new BufferedReader(new InputStreamReader(System.in), 1);
10
11        String line;
12        StringTokenizer words;
13        int count = 0;
14
15        // process lines until no more input
16        while ((line = stdin.readLine()) != null)
17        {
18           // process words in line
19           words = new StringTokenizer(line);
20           while (words.hasMoreTokens())
21           {
22              String word = words.nextToken();
23              boolean isAlphaWord = true;
24              int i = 0;
25
26              // process characters in word
27              while (i < word.length() && isAlphaWord)
28              {
29                 String c = word.substring(i, i + 1).toUpperCase();
30                 if (c.compareTo("A") < 0 || c.compareTo("Z") > 0)
31                    isAlphaWord = false;
32                 i++;
33              }
34              if (isAlphaWord)
35                 count++;
36           }
37        }
38        System.out.println("\n" + count + " alpha words read");
39     }
40  }
```

Figure 2. `CountWords2.java`

A sample dialogue with this program follows:

```
PROMPT>java CountWords2
Maple Leafs 7 Flyers 1
^Z
3 alpha words read
```

Although the inputted line has five tokens, only three are alpha-only. Lines 22-35 perform the added task of determining if each token contains only letters. The first task is to retrieve a token as the `String` object `word` (line 22). Initially, we assume `word` contains only letters, so the `boolean` variable `isAlphaWord` is declared and set to `true` (line 23). Then, each character in `word` is tested (lines 27-33) using the `compareTo()` method of the `String` class. If a character is less than `'A'` or greater than `'Z'`, it is not a letter and `isAlphaWord` is set `false`. This also causes an early termination of the inner `while` loop. If all characters are checked and `isAlphaWord` is still `true`, then the word is, indeed, alpha-only and `count` is incremented (line 35).

Determining if a word contains only letters, as in lines 27-33 of `CountWords2`, uses methods seen in earlier programs. In fact, this task can be simplified:

```
for (int i = 0; i < word.length(); i++)
   if (!Character.isLetter(word.charAt(i)))
      isAlphaWord = false;
```

The work is now performed by the `isLetter()` method of the `Character` wrapper class (see Appendix C). The `isLetter()` method receives a `char` argument and returns a `boolean`: `true` if the character is a letter, `false` otherwise. The `char` argument is obtained via the `charAt()` method of the `String` class. It receives an integer argument specifying the character to extract from the string. Note that `isLetter()` is a class method, whereas `charAt()` is an instance method. Since we are using `isLetter()` to determine if a character is *not* a letter, the unary NOT (`!`) operator precedes the relational expression.

The technique above is not only shorter, it also allows the program to work with the Unicode definition of a letter. Thus, the program will work for non-ASCII interpretations of letters, such as occur in scripts other than English.

Find Palindromes

Now that we are comfortable with the `StringTokenizer` class, let's look at another interesting example. A palindrome is a word that is spelled the same forwards as backwards, for example, "rotator". The program `Palindrome` reads lines of input from the keyboard until an end-of-file condition occurs. As each line is read, it is tokenized and each token is inspected to determine if it is a palindrome. If so, it is echoed to the standard output. The listing is given in Figure 3.

```
 1  import java.io.*;
 2  import java.util.*;
 3
 4  public class Palindrome
 5  {
 6     public static void main(String[] args) throws IOException
 7     {
 8        // open keyboard for input (call it 'stdin')
 9        BufferedReader stdin =
10           new BufferedReader(new InputStreamReader(System.in), 1);
11
12        // prepare to extract words from lines
13        String line;
14        StringTokenizer words;
15        String word;
16
17        // main loop, repeat until no more input
18        while ((line = stdin.readLine()) != null)
19        {
20           // tokenize the line
21           words = new StringTokenizer(line);
22
23           // process each word in the line
24           while (words.hasMoreTokens())
25           {
26              word = words.nextToken();
27              if (word.length() > 2) // must be at least 3 characters long
28              {
29                 boolean isPalindrome = true;
30                 int i = 0;
31                 int j = word.length() - 1;
32                 while (i < j && isPalindrome)
33                 {
34                    if ( !(word.charAt(i) == word.charAt(j)) )
35                       isPalindrome = false;
36                    i++;
37                    j--;
38                 }
39                 if (isPalindrome)
40                    System.out.println(word);
41              }
42           }
43        }
44     }
45  }
```

Figure 3. `Palindrome.java`

A sample dialogue with this program follows:

```
PROMPT>java Palindrome
a deed worth doing
deed
rotator is a palindrome
rotator
^z
```

Two lines of input are entered and these contain two palindromes, "deed" and "rotator". The structure of the program is very similar to that of `CountWords2`. Instead of inspecting each

token to determine if it contains only letters, we check if characters are mirrored about an imaginary center point. The general idea is shown in Figure 4 for "rotator".

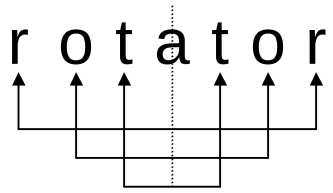


Figure 4. Letter comparisons for palindrome test

Lines 26-40 do the work. First, we set the `boolean` variable `isPalindrome` to `true` with an assumption that the token is a palindrome (line 29). Then, variables `i` and `j` are initialized with the index of the first and last character in the token, respectively (lines 30-31). The `while` loop in lines 32-38 processes pairs of characters starting at the outside, working toward the center. The loop continues only while `i` is less than `j` and `isPalindrome` remains `true`.

The crucial test is in line 34. The `charAt()` method in the `String` class extracts the characters at indices `i` and `j` from the token. These are compared using the "is equal to" relational operator (`==`). Note that `char` variables can be compared as though they were integers. The relational test is inverted with the NOT (`!`) operator. If the overall result is `true`, the characters are *not* equal and the token is *not* a palindrome. In this case, `isPalindrome` is set to `false` (line 35). After each test, `i` is incremented and `j` is decremented, thus moving the indices closer to the center of the token. Tokens with less than two characters are not checked in this program (line 27).

We will re-visit the `Palindrome` program later when we discuss recursion and debugging.

Class Hierarchy - Update

Thus far, we have worked with the `String` class, the `Math` class, the `StringTokenizer`, and Java's eight wrapper classes. Let's pause briefly, and re-visit a topic from earlyr: class hierarchy. We noted that the `Object` class lies at the top of Java's class hierarchy. All classes, either directly or indirectly (through other classes), inherit characteristics from the `Object` class. Earlier, we illustrated the superclass-subclass relationship between the `String` and `Object` classes. Figure 1 expands on this, showing the class hierarchy of all the classes presented thus far.

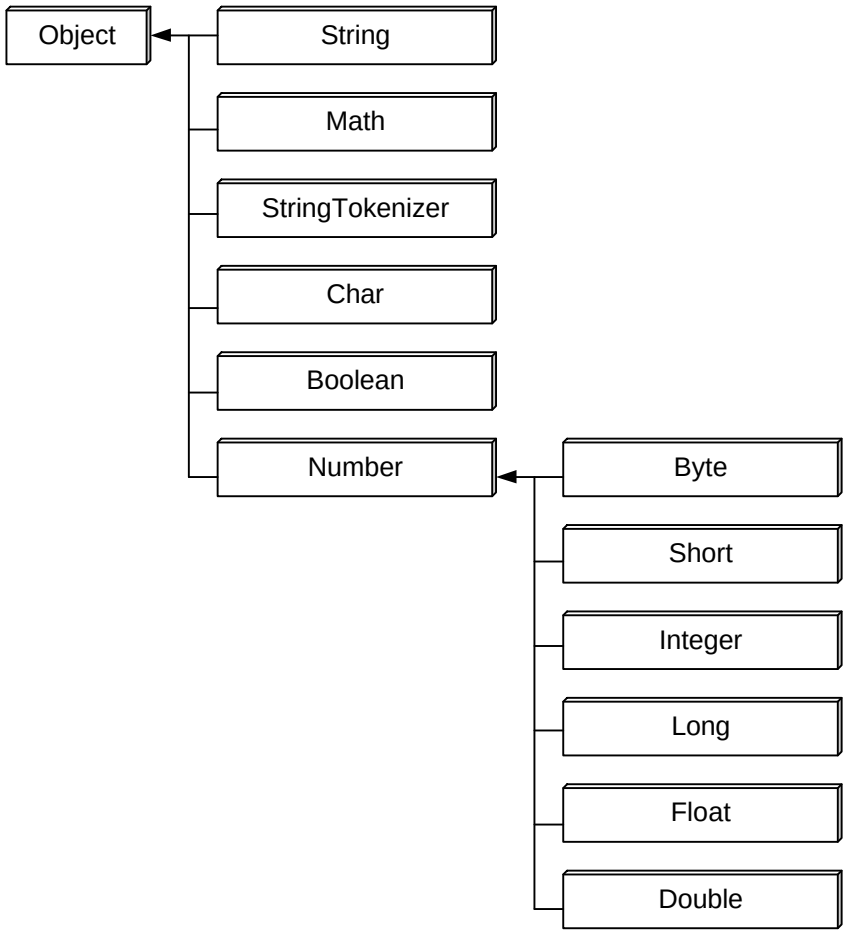


Figure 1. Class hierarchy update

As you might imagine, a complete class hierarchy diagram for all the classes in the Java API could fill a large wall-size poster. (In fact, such posters are available from Java vendors.) A complete diagram would not only be "taller" than Figure 1, it would also be "wider" because many classes are "subclasses of subclasses", and so on. We see an example of this in the six numeric wrapper classes in Figure 1. Each is a subclass of the `Number` class, which, in turn is a subclass of the `Object` class. We will present portions of the class hierarchy on an as-needed basis as we continue our travels through the Java API.

Redirection and Pipes

Providing input via the keyboard to test demo program is tedious. Conveniently, data can be inputted from a file as though they were typed on the keyboard using *redirection*. The "<" input redirection symbol appended to a command line followed by the name of a file forces input to be read from a file instead of from the keyboard. This is purely as system-level service. No modification to the program is required. The following is an example dialogue for the `CountWords2` program using input redirection:

```
PROMPT>java CountWords2 < CountWords2.java
119 alpha words read
```

In this example, the `CountWords2` program is executed in the usual way; however input is redirected. Instead of reading input line-by-line from the keyboard, input is read from the file `CountWords2.java`. (The name of any other text file could be substituted for `CountWords2.java`.) The end-of-input condition generated by entering Control-Z (or Control-D) on the keyboard is generated automatically by the system when the end of the file is reached. Evidently the version of the `CountWords2.java` used for the example above contained 119 tokens containing only letters.

Here is another example using the `Palindrome` program:

```
PROMPT>java Palindrome < Palindrome.java
rotator
```

The only palindrome in the file `Palindrome.java` is the token "rotator". It does not appear in **Error! Reference source not found.** because the comment lines were deleted to keep the listing short.

Output is redirected by appending "`>`" and a filename. Output normally sent to the standard output and displayed on the host system's CRT display is written to the specified file instead.

Input redirection and output redirection apply only to the "standard input" and the "standard output". Reading data from files that are opened and read explicitly in a program is unaffected. Likewise writing data to files that are opened and written explicitly in a program is unaffected. "Explicitly" opening and reading or writing data files is discussed later. Just think of input redirection as a convenient way to input data from a file instead of entering it on the keyboard. Output redirection is just a convenient way to save data in a file that are normally sent to the host system's display.

A close cousin to input/output redirection is a *pipe*. A pipe provides a link between two programs. The output of the first is presented as input to the second. As with redirection, a pipe affects only the standard output and the standard input. The pipe symbol is " | ".

With redirection and pipes, we are in a position to perform substantial processing on large amounts of data — provided the data are stored in a text file. A text file contains lines of "displayable" characters separated by newline characters. This is in contrast to binary files that contain non-displayable characters representing computer instructions, formatting codes, etc. To demonstrate redirection and pipes, let's examine two more Java programs.

Echo Words

An interesting variation of the `CountWords2` programs is shown in Figure 1. The program `EchoAlphaWords` reads lines from the standard input, tokenizes each line, and echoes to the standard output each token that contains only letters.

```
 1  import java.io.*;
 2  import java.util.*;
 3
 4  public class EchoAlphaWords
 5  {
 6     public static void main(String[] args) throws IOException
 7     {
 8        // prepare keyboard for input
 9        BufferedReader stdin =
10              new BufferedReader(new InputStreamReader(System.in), 1);
11
12        // prepare to extract words from a line
13        StringTokenizer words;
14        String line;
15        String word;
16
17        // process lines until 'null' (no more input)
18        while ((line = stdin.readLine()) != null)
19        {
20           // prepare to tokenize line
21           words = new StringTokenizer(line);
22
23           // process words in line
24           while (words.hasMoreTokens())
25           {
26              word = words.nextToken();
27              boolean isAlphaWord = true;
28
29              // process characters in word
30              for (int i = 0; i < word.length(); i++)
31                 if (!Character.isLetter(word.charAt(i)))
32                    isAlphaWord = false;
33              if (isAlphaWord)
34                 System.out.println(word);
35           }
36        }
37     }
38  }
```

Figure 1. `EchoAlphaWords.java`

`EchoAlphaWords` is very similar to the `CountWords` presented earlier. The main difference is that it sends words to the standard output (see line 34, Figure 1) rather than simply counting words (see line 23, **Error! Reference source not found.**). A sample dialogue with `EchoAlphaWords` follows:

```
PROMPT>java EchoAlphaWords
java is fun
java
is
fun
^z
```

Of course, `EchoAlphaWords` can be used with input redirection. For example,

```
PROMPT>java EchoAlphaWords < CountWords2.java
```

echoes all words in the `CountWords2.java` file to the standard output, one word per line. The output is shown Figure 2. To conserve space, the output is shown across four columns. Read down the first column, then left to right. Note that the file contains words in comment lines that do not appear in **Error! Reference source not found.**.

program	to	demonstrate	loops
within	loops	Same	as
CountWords	except	only	words
containing	letters	of	the
alphabet	are	There	are
three	levels	of	outer
loop	read	lines	of
input	until	no	more
input	middle	loop	process
words	in	a	line
inner	loop	process	characters
in	a	word	A
useful	source	of	input
is	a	disk	file
which	may	be	read
using	input	A	sample
dialog	is	alpha	words
read	Scott	import	import
public	class	public	static
void	throws	IOException	BufferedReader
stdin	new	String	StringTokenizer
int	count	process	lines
until	no	more	input
while	process	words	in
line	words	new	while
String	word	boolean	isAlphaWord
int	i	process	characters
in	word	while	String
c	i	if	isAlphaWord
if	alpha	words	

Figure 2. Alpha words in `CountWords2.java`

Let's develop this idea further. Suppose we want to count the number of *unique* alpha-only words in a file. We're not quite there yet, because `EchoAlphaWords` echoes words in the order they appear, so the output may contain multiple instances of the same word. There is a DOS utility called `sort` that can help. By default, `sort` reads the standard input and sorts lines in ascending order. Since `EchoAlphaWords` outputs one word per line, "piping" the output of `EchoAlphaWords` through `sort` generates a sorted list of words. Here is a simple dialogue without using input redirection:

```
PROMPT>java EchoAlphaWords | sort
programming in java is fun
^z
fun
in
is
java
programming
```

One line of input containing five words was entered. The line was tokenized and the tokens were sent to the standard output, one token per line. However, the output was "caught" by the pipe and sent as input to the `sort` utility. The output of the `sort` utility is the same five tokens, sorted in ascending order.

A sorted list of the words in the file `CountWords2.java` is generated as follows:

```
PROMPT>java EchoAlphaWords < CountWords2.java | sort
```

Lots of words are repeated. For example, the word "while" appears three times in the output. Before we can count unique words we need to remove duplicate words. The program `FilterDuplicateWords` does this task (see Figure 3).

```
 1  import java.io.*;
 2  import java.util.*;
 3
 4  public class FilterDuplicateWords
 5  {
 6     public static void main(String[] args) throws IOException
 7     {
 8        // prepare keyboard for input
 9        BufferedReader stdin =
10              new BufferedReader(new InputStreamReader(System.in), 1);
11
12        // prepare to extract words from a line
13        StringTokenizer words;
14        String line;
15        String word;
16        String oldWord = "";
17
18        // process lines until 'null' (no more input)
19        while ((line = stdin.readLine()) != null)
20        {
21           // prepare to tokenize line
22           words = new StringTokenizer(line);
23
24           // process words in line
25           while (words.hasMoreTokens())
26           {
27              word = words.nextToken();
28              boolean isAlphaWord = true;
29              int i = 0;
30              if (!oldWord.equals(word))
31              {
32                 oldWord = word;
33                 System.out.println(word);
34              }
35           }
36        }
37     }
38  }
```

Figure 3. `FilterDuplicateWords.java`

This program keeps a temporary copy of each word in the `String` object `oldWord` (line 32). As each new word is read, it is sent to the standard output only if it differs from the previous word (lines 30-34).

With `FilterDuplicateWords` added to our trick bag, we are ready to count the unique words in the file `CountWords2.java`. The following command line performs this task:

```
PROMPT>java EchoAlphaWords < CountWords2.java | sort | java
    FilterDuplicateWords | java CountWords2
48 alpha words read
```

The command overflows to a second line, but it is processed as a single command. Three pipes are used in combining three Java programs with a DOS utility. Of the 119 alpha words in `CountWords2.java`, 48 are unique.

Stream Classes and File I/O

A *stream* is any input source or output destination for data. The Java API includes about fifty classes for managing input/output streams. Objects of these classes can be instantiated and methods exist to perform "actions" on these objects. The most basic actions are sending data to a stream or receiving data from a stream. In this section, we'll examine the basic operation of streams, focusing on the standard I/O streams and on file streams. For a detailed look at Java's stream classes, see Chapter 12 in Arnold and Gosling's *The Java Programming Language*.

A Java program is reasonably well served by its default state when execution begins. Three streams are setup and ready to go. There are two output streams, identified by the objects `System.out` and `System.err`, and one input stream, identified by the object `System.in`. These objects are defined as public data fields in the `System` class of the `java.lang` package (see Table 1).

Table 1. Data Fields in the `System` Class

Variable	Type	Purpose
`err`	`PrintStream`	the "standard error" output stream. This stream is already open and ready to accept output data.
`in`	`InputStream`	the "standard input" stream. This stream is already open and ready to supply input data.
`out`	`PrintStream`	the "standard output" stream. This stream is already open and ready to accept output data.

As noted in the table, the streams are "open and ready" when a Java program begins execution. To use these objects in a program, they must be prefixed with "`System.`" to identify where they are defined. This is similar to our use of the prefix "`Math.`" when using methods of the `Math` class.

The `err` and `out` objects are instances of the `PrintStream` class and the `in` object is an instance of the `InputStream` class. As you might guess, methods in these classes provide the means to input or output data. As well, the default devices for these streams are the keyboard for input and the host system's CRT display for output.

Two output streams are provided to separate "normal" output from error messages. Normal output can be redirected to a file, as demonstrated earlier, and this is a useful service. However, if errors occur during program execution, error messages appear on the host system's CRT display, even if output was redirected to a file. In most cases, it is useful to separate these two forms of output, and, so, "standard output" and "standard error" streams exist.

Let's begin by examining the standard output stream.

The Standard Output Stream

We have already used the standard output stream extensively. Variations of the following statement

```
System.out.println("Hello");
```

appear frequently in our demo programs. The method `println()` is called via an instance of the `PrintStream` class, in this case the object variable `out`. The methods of the `PrintStream` class are summarized in Table 2.

Table 2. Methods of the `PrintStream` Class

Method	Purpose
Constructors	
`PrintStream(OutputStream out)`	creates a new print stream. This stream will not flush automatically; returns a reference to the new `PrintStream` object
`PrintStream(OutputStream out, boolean autoFlush)`	creates a new print stream. If `autoFlush` is true, the output buffer is flushed whenever (a) a byte array is written, (b) one of the `println()` methods is invoked, or (c) a newline character or byte ('\n') is written; returns a reference to the new `PrintStream` object
Instance Methods	
`flush()`	flushes the stream; returns a `void`
`print(char c)`	prints a character; returns a `void`
`println()`	terminates the current line by writing the line separator string; returns a `void`
`println(char c)`	prints a character and then terminates the line; returns a `void`

Note that the `print()` and `println()` methods are overloaded. They can accept any of the primitive data types as arguments, as well as character arrays or strings.

If `println()` is called without an argument it serves only to terminate the current line by outputting a newline character ('`\n`'). The method `print()` outputs its argument as a series of characters without automatically sending a newline character. On some systems, the `print()`must be followed by the `flush()` to force characters to print immediately. Otherwise, they are held in an output buffer until a subsequent newline character is printed. This behaviour is established when the output stream is opened, as seen in the descriptions of the constructor methods. The example programs in this text were written using *Java 2* on a host system running the *Windows98* operating system. The `autoFlush` option is "on", and, so the `flush()` method is not needed. It may be necessary to follow `print()` with `flush()` on other systems.

As demonstrated in earlier, the standard output stream may be redirected to a disk file by appending "`>`" followed by a filename when the program is invoked.

The Standard Input Stream

Unfortunately, the behaviour of Java programs is not quite as clean for the standard input stream as it is for the standard output stream. The object variable "`in`" is defined in the `System` class as an instance of the `InputStream` class. As is, the standard input stream is setup to read raw, unbuffered bytes from the keyboard. Such low-level control may be necessary in some cases, but

the result is inefficient and provides poor formatting of input data. For example, it forces us to deal explicitly with keystrokes such as "backspace" and "delete". For most purposes, this is a nuisance. Earlier, we illustrated a convenient way to re-package the standard input stream:

```
BufferedReader stdin =
   new BufferedReader(new InputStreamReader(System.in), 1);
```

The object `System.in` appears above as an argument to the constructor method for the `InputStreamReader` class. The resulting object — an instance of the `InputStreamReader` class — in turn appears as an argument to the constructor method for the `BufferedReader` class. A reference to the instantiated object is assigned to the object variable `stdin`.

There is a long story and a short story to the above syntax. If you want all the details on the `InputStream` class and the `InputStreamReader` class, see the Java API documentation. The short story is that our access to the standard input stream now occurs via the object variable `stdin` which is an instance of the `BufferedReader` class in the `java.io` package. So, the methods of the `BufferedReader` class are of primary concern to us. These are summarized in Table 3.

Table 3. Methods of the `BufferedReader` Class

Methods	Description
Constructor Methods	
`BufferedReader(Reader in)`	creates a buffered character-input stream that uses a default-sized input buffer; returns a reference to the new object
`BufferedReader(Reader in, int sz)`	creates a buffered character-input stream that uses an input buffer of the specified size; returns a reference to the new object
Instance Methods	
`close()`	returns a `void`; closes the stream
`readLine()`	returns a `String` object equal to a line of text read

For our purposes, we only need the `readLine()` method. Each line typed on the keyboard is returned as a `String` object. For example,

```
String s = stdin.readLine();
```

If the line contains a representation of a primitive data type, it is subsequently converted using a "parsing" method of a wrapper class. If an integer value is input from the keyboard, the inputed string is converted to an `int` as follows:

```
int i = Integer.parseInt(s);
```

Since the `String` variable `s` isn't needed, the two statements above can be combined:

```
int i = Integer.parseInt(stdin.readLine());
```

If the line contains multiple words, or tokens, they are easily separated using the `StringTokenizer` class. This was demonstrated earlier.

We will use the `close()` method later when we learn to access disk files for input using the `BufferedClass`. The standard input stream is automatically closed upon program termination.

File Streams

The streams used most often are the standard input (the keyboard) and the standard output (the CRT display). Alternatively, standard input can arrive from a disk file using "input redirection", and standard output can be written to a disk file using "output redirection". (Redirection was discussed earlier.) Although I/O redirection is convenient, there are limitations. For one, redirection is "all or nothing". It is not possible, for example, to read data from a file using input redirection *and* receive user input from the keyboard. As well, it is not possible to read or write multiple files using input redirection. A more flexible mechanism to read or write disk files is available through Java's *file streams*.

The two file streams presented here are the *file reader stream* and the *file writer stream*. As with the standard I/O streams, we access these through objects of the associated class, namely the `FileReader` class and the `FileWriter` class. Unlike the standard I/O streams, however, we must explicitly "open" a file stream before using it. After using a file stream, it is good practice to "close" the stream, although, strictly speaking, this is not necessary as all streams are automatically closed when a program terminates.

Reading Files

Let's begin with the `FileReader` class. As with keyboard input, it is most efficient to work through the `BufferedReader` class. If we wish to read text from a file named `foo.txt`, it is opened as a file input stream as follows:

```
BufferedReader inputFile =
   new BufferedReader(new FileReader("foo.txt"));
```

The line above "opens" `foo.txt` as a `FileReader` object and passes it to the constructor of the `BufferedReader` class. The result is a `BufferedReader` object named `inputFile`. We have used the `BufferedReader` class extensively already for keyboard input. Not surprisingly, the same methods are available for `inputFile` as for `stdin` in our earlier programs. To read a line of text from `junk.txt`, we use the `readLine()` method of the `BufferedReaderClass`:

```
String s = inputFile.readLine();
```

Note that `foo.txt` is *not* being read using input redirection. We have explicitly opened a file input stream. This means the keyboard is still available for input. So, for example, we could prompt the user for the name of a file, instead of "hard coding" the name in the program:

```
// get the name of a file to read
System.out.print("What file do you want to read: ");
String fileName = stdin.readLine();

// open the file for reading
BufferedReader inputFile =
   new BufferedReader(new FileReader(filename));

// read the file
String s = inputFile.readLine();
```

Note, above, that keyboard input *and* file input take place in the same program. Once finished with the file, the file stream is closed as follows:

```
inputFile.close();
```

Some additional file I/O services are available through Java's `File` class, which supports simple operations with filenames and paths. For example, if `fileName` is a string containing the name of a file, the following code checks if the file exists and, if so, proceeds only if the user enters "y" to continue.

```
File f = new File(fileName);
if (f.exists())
{
   System.out.print("File already exists. Overwrite (y/n)? ");
   if(!stdin.readLine().toLowerCase().equals("y"))
      return;
}
```

Let's move on to an example. Figure 1 shows a simple Java program to open a text file called `temp.txt` and to count the number of lines and characters in the file.

```
 1  import java.io.*;
 2
 3  public class FileStats
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        // the file must be called 'temp.txt'
 8        String s = "temp.txt";
 9
10        // see if file exists
11        File f = new File(s);
12        if (!f.exists())
13        {
14           System.out.println("\'" + s + "\' does not exit. Bye!");
15           return;
16        }
17
18        // open disk file for input
19        BufferedReader inputFile =
20           new BufferedReader(new FileReader(s));
21
22        // read lines from the disk file, compute stats
23        String line;
24        int nLines = 0;
25        int nCharacters = 0;
26        while ((line = inputFile.readLine()) != null)
27        {
28           nLines++;
29           nCharacters += line.length();
30        }
31
32        // output file statistics
33        System.out.println("File statistics for \'" + s + "\'...");
34        System.out.println("Number of lines = " + nLines);
35        System.out.println("Number of characters = " + nCharacters);
36
37        // close disk file
38        inputFile.close();
39     }
40  }
```

Figure 1. `FileStats.java`

A sample dialogue with the program follows (user input is underlined):

```
PROMPT>java FileStats.java
File statistics for 'temp.txt'...
Number of lines = 48
Number of characters = 1270
```

Here's a programming challenge for you: Make a copy of `FileStats.java` and name the new file `FileStats2.java`. Modify the program so that the user is prompted to enter the name of the file to open. This makes more sense than to hard code the file's name within the program.

Writing Files

To open a file output stream to which text can be written, we use the `FileWriter` class. As always, it is best to buffer the output. The following sets up a buffered file writer stream named `outFile` to write text into a file named `save.txt`:

```
PrintWriter outFile =
   new PrintWriter(new BufferedWriter(new FileWriter("save.txt"));
```

Once again, we have packaged the stream to behave like one Java's standard streams. The object `outFile`, above, is of the `PrintWriter` class, just like `System.out`. If a string, `s`, contains some text, it is written to the file as follows:

```
outFile.println(s);
```

When finished, the file is closed as expected:

```
outFile.close();
```

Reading and write disk files, as just described, is plain and simple. We use the same methods as for reading the keyboard (the standard input) or for writing to the CRT display (the standard output). Of course, situations may arise with disk files that are distinctly different, and these should be anticipated and dealt with as appropriate. Earlier, we used the `exists()` method of the `File` class to ckeck if a file exists before attempting to open it, and the same approach may be used here. However, if we open and write data to an existing file, what happens to the previous contents? The old file is truncated and the previous contents are overwritten. In many situations, this is undesirable, and a variety of alternative approaches may be taken. The `FileWriter` constructor can be used with two arguments, where the second is a `boolean` argument specifying an "append" option. For example, the expression

```
new FileWriter("save.txt", true)
```

opens `save.txt` as a file output stream. If the file currently exists, subsequent output is appended to the file.

Another possibility is opening an existing *read-only* file for writing! In this case, the program terminates with an "access is denied" exception occurs. This can be caught and dealt with in the program, and we'll learn how to do so later.

Figure 2 shows a simple Java program to read text from the keyboard and write the text to a file called `junk.txt`.

```
 1  import java.io.*;
 2
 3  class FileWrite
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7
 8        // open keyboard for input (call it 'stdin')
 9        BufferedReader stdin =
10           new BufferedReader(new InputStreamReader(System.in));
11
12        // Let's call the output file 'junk.txt'
13        String s = "junk.txt";
14
15        // check if output file exists
16        File f = new File(s);
17        if (f.exists())
18        {
19           System.out.print("Overwrite " + s + " (y/n)? ");
20           if(!stdin.readLine().toLowerCase().equals("y"))
21              return;
22        }
23
24        // open file for output
25        PrintWriter outFile =
26           new PrintWriter(new BufferedWriter(new FileWriter(s)));
27
28        // inform the user what to do
29        System.out.println("Enter some text on the keyboard...");
30        System.out.println("(^z to terminate)");
31
32        // read from keyboard, write to file output stream
33        String s2;
34        while ((s2 = stdin.readLine()) != null)
35           outFile.println(s2);
36
37        // close disk file
38        outFile.close();
39     }
40  }
```

Figure 2. `FileWrite.java`

A sample dialogue with this file follows (user input is underlined):

```
PROMPT>java FileWrite
Enter some text on the keyboard...
(^z to terminate)
Happy new year!
^z
PROMPT>
```

Note that `^z` means hold the Control key down while pressing `z`. (On unix systems, `^d` must be entered.)

If the program is immiately run again, then the following dialogue results:

```
PROMPT>java FileWrite
Overwrite junk.txt (y/n)? y
Enter some text on the keyboard...
(^z to terminate)
Happy birthday!
^z
PROMPT>
```

Since the file `junk.txt` was created on the first pass, it already exists on the second. Our code (see lines 15-22) provides a reasonable safety check before proceeding. Since the user entered "y", the program proceeds; however, the previous contents are overwritten.

There are several ways to demonstrate the "exceptional conditions" that may arise when working with disk files, and these are worth considering. Once the file `junk.txt` exists, make it "read only" and re-run `FileWrite` The program will terminate with the "access is denied" message noted earlier.[1] Another interesting demonstration is to open `junk.txt` in an editor, and then exectute `FileWrite`. The behaviour may vary from one editor to the next, but we'll leave this for you to explore.

Here's a programming challenge for you: Make a copy of `FileWrite.java` and name the new file `FileWrite2.java`. Modify the program such that `junk.txt` is opened in append mode. Compile the new program, then run it a few times to convince yourself that the file contents are preserved from one invocation to the next.

Class Hierarchy - Stream Classes

Earlier, we presented a class hierarchy showing Java's eight wrapper classes, the `Math` class, the `String` class, and the `StringTokenizer` class. In our present discussion of stream classes, several classes have been mentioned, and, of course, each holds a position in Java's overall class hierarchy. This is illustrated Figure 3. A few additional classes are shown that were not discussed. (You may want to explore these on your own.)

[1] To make `junk.txt` "read only", do the following. On *Windows* systems, locate the file in Windows Explorer then right-click on it and tick the "read-only" box. On unix systems, enter `chmod 444 junk.txt`.

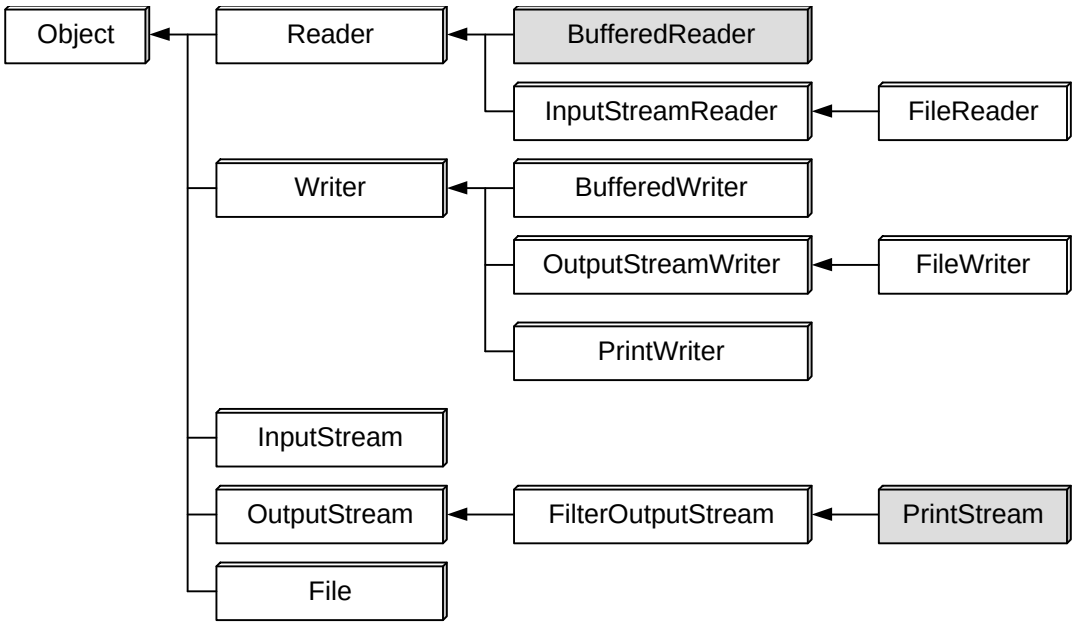


Figure 3. Class hierarchy for stream classes

The objects used in our demo programs for keyboard input and for output to the host systems' CRT are `stdin` and `System.out`. `stdin` is an object of the `BufferedReader` class and `System.out` is an object of the `PrintStream` class. These classes are highlighted in Figure 3. As we are well aware, instance methods of a class are accessed via instances variables of that class. So, for example the instance method `println()` of the `PrintStream` class can be accessed, or called, through `System.out`. As well, instance methods of a superclass may be accessed by an object of a subclass. (Recall that a subclass inherits characteristics, such as methods, from its superclass). So, for example, instance methods of the `PrintStream`, `FilterOutputStream`, `OutputStream`, and `Object` classes are all accessible via `System.out`, because `PrintStream` inherits from `FilterOutputStream`, which inherits from `OutputStream`, which, in turn, inherits from `Object`.

Now, if you are inclined to search through the Java API documentation to see what methods are available and to write small programs to make methods "further up" act on objects "further down", congratulations: You have caught the "Java bug" — not in the programming-error sense, but in the Java-is-fun sense. And, by all means, please do. The main point here, however, is to alert you to — and keep you thinking about — the overall structure and organization of the Java API.

The System Class

We have already met the all-important data fields defined in the `System` class to setup the standard input and standard output streams. The `System` class also includes a variety of special-purpose methods (see the Java API documentation); however, we will only discuss one here. The `currentTimeMillis()` method returns a `long` equal to the current time in milliseconds (ms). Note that 1 ms equals 10^{-3} seconds. The value returned is relative to the beginning of January 1, 1970; so, in most practical applications, it is the difference between two successive calls to `currentTimeMillis()` that is of interest. Java includes a comprehensive set of classes for dates and times, and we'll explore these later. For the moment, let's see the possibilities for the `currentTimeMillis()` method in the `System` class. A simple "seconds counter" is demonstrated in the program `CountSeconds` (see Figure 1).

```
 1  public class CountSeconds
 2  {
 3     public static void main(String[] args)
 4     {
 5        long start = System.currentTimeMillis();
 6        long seconds = 0;
 7        while(true)
 8        {
 9           long temp = System.currentTimeMillis();
10           long newSeconds = (temp - start) / 1000;
11           if (newSeconds != seconds)
12           {
13              System.out.print(newSeconds + " ");
14              seconds = newSeconds;
15           }
16        }
17     }
18  }
```

Figure 1. `CountSeconds.java`

This program generates the following output:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 etc.
```

Each number is printed "precisely" one second following the previous one. The timing of the output is as precise as the host system's clock.

The main loop (lines 7-16) is setup with "`while (true)`", so it is an infinite loop. The program executes indefinitely and continues to count seconds until the user enters Control-c to terminate the program. The `currentTimeMillis()` method is first called in line 5. The value read is a very large integer since it equals the number of milliseconds since January 1, 1970. It's not much use, as is. Nevertheless, it is assigned to the `long` variable `start`. In the next line, a `long` variable `seconds` is declared and initialized with zero.

Within the `while` loop, `currentTimeMillis()` is called again (line 9), with the new value assigned to the `long` variable `temp` (line 9). The original value in `start` is subtracted from `temp` and the result is divided by 1000. This result, which equals the elapsed time in seconds between the two calls, is assigned to the long variable `newSeconds` (line 10). (Note: the result of the expression "`(temp - start) / 100`" yields a `long` integer.)

All the tricky stuff is in the `if` statement in lines 11-15. First, the `long` variables `newSeconds` and `seconds` are compared. If they are not equal, the statement block is skipped and the `while` loop begins again with a new call to `currentTimeMillis()`. Only after 1000 ms (1 second) has elapsed between the first two calls to `currentTimeMillis()` will `newSeconds` finally equal 1. At this point, the relational test in the `if` statement passes and the statement block is executed (lines 12-15). The value of `newSeconds` is printed (It equals "1" the first time through!), and then `seconds` is reassigned to `newSeconds` (line 14). This last step is crucial, as it ensures the relational expression in the `if` statement fails again, repeatedly, until another 1000 ms has elapsed.

The `while` loop in `CountSeconds` executes "many" times for each value printed. The actual number varies on the speed of the host system, and on the number and activity of background processes. To investigate this, `CountSeconds` was re-worked with a few modifications (see Figure 2).

```
 1  public class CountSeconds2
 2  {
 3     public static void main(String[] args)
 4     {
 5        long start = System.currentTimeMillis();
 6        long seconds = 0;
 7        int i = 0;
 8        while(true)
 9        {
10           i++;
11           long temp = System.currentTimeMillis();
12           long newSeconds = (temp - start) / 1000;
13           if (newSeconds != seconds)
14           {
15              System.out.println(i);
16              i = 0;
17              seconds = newSeconds;
18           }
19        }
20     }
21  }
```

Figure 2. `CountSeconds2.java`

`CountSeconds2` leaves in tact most of the activities in `CountSeconds`. However, a new variable is introduced. The `int` variable `i` is declared and initialized to zero in line 7. Each time through the `while` loop, `i` is incremented (line 10); however, it is only printed once per second, inside the `if` statement (line 15). After `i` is printed, it is reset to zero (line 16). A sample ten-second run of this program is shown below.

```
6071
4744
4173
5459
5470
5762
5450
5476
5463
5411
```

So, the `while` loop executes approximately 5400 times each second. The high count was 6071, the low count was 4173. The variation is due to background processing on the host system. This output was generated from a notebook computer with a 166 MHz Pentium MMX processor running the *Windows98* operating system. The counts above will differ for machines with faster or slower CPU clock speeds.

The StringBuffer Class

We noted earlier that `String` objects are "immutable" in Java. Once a `String` object is instantiated, it cannot change in size or content. Any change yields a new `String` object and the old one is discarded. Strings created with the `StringBuffer` class, however, are dynamic. Once created, characters can change and new characters can be inserted or deleted. Although these tasks are possible through the creative use of substringing and concatenation with `String` objects, there are performance benefits in using `StringBuffer` objects when such manipulations are frequent.

The `StringBuffer` class is part of the `java.lang` package. The most common methods are summarized in Table 1.

Table 1. Methods of the `StringBuffer` Class

Method	Description
Constructors	
`StringBuffer()`	constructs a `StringBuffer` object with no characters in it and an initial capacity of 16 characters; returns a reference to the new object
`StringBuffer(int length)`	constructs a `StringBuffer` object with no characters in it and an initial capacity specified by `length`; returns a reference to the new object
`StringBuffer(String s)`	constructs a `StringBuffer` object so that it represents the same sequence of characters as the string `s`; returns a reference to the new object
Instance Methods	
`append(char c)`	appends the character to the string buffer; returns a reference to this `StringBuffer`
`charAt(int i)`	returns the `char` at the specified index in the string buffer
`delete(int i, int j)`	deletes characters from positions `i` to `j - 1`; returns a reference to this `StringBuffer`
`deleteCharAt(int i)`	deletes the character at position `i`; returns a reference to this `StringBuffer`
`insert(int i, char c)`	inserts a character at position `i`; returns a reference to this `StingBuffer`
`length()`	returns and `int` equal to the length (character count) of the string buffer
`replace(int i, int j, String s)`	removes characters from position `i` to `j - 1`, then inserts string `s`; returns a reference to this `StringBuffer`
`reverse()`	reverses the characters in the string buffer; returns a reference to this `StringBuffer`
`setCharAt(int i, char c)`	sets the character at position `i` to the character `c`; returns a reference to this `StringBuffer`
`substring(int i)`	returns a `String` object equal to a substring of the string buffer from position `i` to the end of the string buffer
`toString()`	returns a `String` object equivalent to this `StringBuffer`

Note that the `append()` and `insert()` methods are overloaded. They can accept any primitive data type as an argument, as well as character arrays or strings.

The compiler uses methods of the `StringBuffer` class to implement the concatenation operation for `String` objects, but this is transparent to programmers. As an example, the following statement

```
String s = "Y" + 2 + "K";
```

is compiled to the equivalent of

```
String s = new StringBuffer().append("Y").append(2).append("K).toString();
```

which creates a new string buffer (initially empty), and appends in turn the string representation of each operand. Then, the content of the string buffer is converted to a `String` object. Overall, this avoids creating many temporary strings.

Every string buffer has a capacity. As long as the number of characters does not exceed its capacity, all is well. However, if the internal buffer overflows due to an `append()` or `insert()` method, it is automatically made larger. This occurs in the background, and is transparent to the programmer or user.

A demo program using the `StringBuffer` class is presented in the next section.

The Random Class

Random numbers are extremely useful, for example, in generating moves in a game or as test data for computer programs. If one is asked to "pick a number between one and a hundred", the task seems simple enough. But, if you need truly random numbers (each number is equally probable!), and if you want to generate them from a computer, the task is quite tricky as it turns out. Mathematicians have laboured over this problem, and have devised robust techniques to generate random numbers. However, computers are deterministic (all actions are predictable — at some level!), and, so, generating numbers that are "truly" random is not possible. However, we can get pretty close. Algorithms that generate random numbers, admittedly, provide "pseudo-random" numbers. But, for most purposes this is good enough.

Java's `Random` class in the `java.util` package includes a variety of methods to generate pseudo-random numbers. These are summarized in Table 1.

Table 1. Methods of the `Random` Class

Methods	Decsription
Constructor	
`Random()`	creates a new random number generator; returns a reference to the new object
Instance Methods	
`nextBoolean()`	returns a `boolean`: the next pseudo-random, uniformly distributed `boolean` value from this random number generator's sequence
`nextDouble()`	returns a `double`: the next pseudo-random, uniformly distributed `double` value between 0.0 and 1.0 from this random number generator's sequence
`nextGaussian()`	returns a `double`: the next pseudo-random, Gaussian ("normally") distributed `double` value with mean 0.0 and standard deviation 1.0 from this random number generator's sequence
`nextInt()`	returns an `int`: the next pseudo-random, uniformly distributed `int` value from this random number generator's sequence
`nextInt(int n)`	returns an `int`: a pseudo-random, uniformly distributed `int` value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.

Example: Random Bits

Let's put the `Random` class to work in a demo program. The `RandomBits` program in Figure 1 generates a series of random bits (0 or 1). Each bit is represented as a character and placed in a 10-character string buffer (initially containing only spaces) at a random location in the string. After each bit is inserted, the string buffer is printed. A delay of 100 ms between print statements improves the visual output.

```
 1  import java.util.*;
 2
 3  public class RandomBits
 4  {
 5     public static void main(String[] args)
 6     {
 7        long t1 = System.currentTimeMillis();
 8        int temp = 0;
 9        StringBuffer sb = new StringBuffer("          ");  // 10 spaces
10        Random r = new Random();
11        while(true)
12        {
13           int t2 = (int)(System.currentTimeMillis() - t1) / 100;
14           if (t2 != temp)
15           {
16              int i = r.nextInt(10);
17              String bit = r.nextBoolean() ? "1" : "0";
18              sb.replace(i, i + 1, bit);
19              System.out.println(sb);
20              temp = t2;
21           }
22        }
23     }
24  }
```

Figure 1. `RandomBits.java`

A sample 10-second run of `RandomBits` is shown in Figure 2. To conserve space, the 100 lines of output are shown across four columns.

```
   0          0011010101    1010011001    1100011010
   0  0       0011010100    1010011011    1100011010
   0  0   1   0011010100    1010011011    1110011010
   0  0   1   0011010100    1010011011    1110011000
   1  0   1   0010010100    1010011010    1110111000
   1  00  1   1010010100    1010011010    1110111000
   1  10  1   1011010100    1010011010    1110111000
   1  11  1   1011010101    1010011010    1110111010
  0 1  11  1   1011010101    1010011010    1110111010
  0 1 011  1   0011010101    1000011010    1111111010
  001 011  1   0011010101    1000011010    1111111010
  0011011  1   0011010101    1000001010    1111111010
  0111011  1   0010010101    1000001010    1101111010
  0111011  0   0010010111    1000001011    1100111010
  0111011  0   0010011111    1000001011    1100111010
  0111011  1   0010011111    1000001011    1100111010
  0111011  0   0010011111    1000001010    1110111010
  01110111 0   0010011101    1000001010    1110011010
  01100111 0   0010011101    1000001010    1010011010
  0110011100   0010011111    1000001000    1010001010
  0010011100   0010011101    1000001010    0010001010
  0010011100   0010011101    1000001010    0010001110
  0010010100   1010011101    1100001010    0110001110
  0010010100   1010011101    1100001010    0110001110
  0011010100   1010011101    1100101010    0110000110
  0011010101   1010011001    1100111010    0100000110
```

Figure 2. Sample output from `RandomBits.java`

This program is little more than pesky fun; however it brings together several methods and classes. The `currentTimeMillis()` method of the `System` class is used in creating the 100 ms delay between successive print statements. The `replace()` method of the `StringBuffer` class is used to place one-character strings ("0" or "1") in a dynamic string. The `nextInt()` and `nextBoolean()` methods of the `Random` class provide random indices and random bits.

The implementation of the 100 ms time delay is very similar to the delay demonstrated earlier in `CountSeconds.java`, so we needn't say more here. The act of placing random characters in a string at random locations in the string requires two random numbers. The string contains ten characters, so the indices vary from 0 to 9. To generate a random index between 0 and 9, the `nextInt()` method is called with "10" as an argument (line 16). A pseudo-random integer between 0 and 9 is returned and assigned to the `int` variable `i`. In line 17, a pseudo-random bit is generated using the `nextBoolean()` method. Since `nextBoolean()` returns a `boolean` (`true` or `false`) the selection operation helps convert the result to the string "0" or the string "1". Recall that the statement

```
String bit = r.nextBoolean() ? "1" : "0";
```

is equivalent to

```
String bit;
if (r.nextBoolean())
   bit = "1";
else
   bit = "0";
```

Note that a relational test is always a test for "`true`"; so, the expression

```
r.nextBoolean()
```

is equivalent to

```
r.nextBoolean() == true
```

A `StringBuffer` object `sb` is instantiated in line 9 and initialized with ten space characters. One character is replaced with "0" or "1" in line 18, and the new string in printed in line 19. The critical step occurs in line 18 with the `replace()` method. The method replaces strings; so the arguments are the one-character `String` object `bit` and the indices `i` and `i + 1`.

Note in the sample output (Figure 2) that both the value replaced and the position appear random from one line to the next. At most, one character changes. In some cases, no character changes. This occurs approximately half the time — when the new random bit happens to match the existing bit.

The `Random` class documentation in the Java API documentation, includes details on the algorithms for generating random numbers. Consult this for more details or see references [4] or [5].

If all you want is a random `double`, as provided by the `nextDouble()` method, you can avoid instantiating a `Random` object. The `Math` class of the `java.lang` package includes a method called `random()` with the same effect. Once again, behind-the-scenes activities do the work and hide the details. The statement

```
double x = Math.random();
```

first creates a single new pseudo-random number generator, exactly as if by the expression

```
new java.util.Random()
```

and then retrieves a random `double` as if by the expression

```
nextDouble()
```

The new pseudo-random number generator is used thereafter for all calls to `random()`.

Example: Random Quilt

The next example is an applet using the `nextDouble()` method of the `Math` class. `RandomQuilt` randomly generates a quilt-like pattern of colored patches in a graphics window. The pattern changes ten times per second. The source code is shown in Figure 3.

```
 1  import java.awt.*;
 2  import java.applet.*;
 3
 4  public class RandomQuilt extends Applet
 5  {
 6     public void paint(Graphics g)
 7     {
 8        long t1 = System.currentTimeMillis();
 9        int temp = 0;
10        while (true)
11        {
12           // updates 10 times per seconds
13           int t2 = (int)(System.currentTimeMillis() - t1) / 100;
14           if (t2 != temp)
15           {
16              // generate arguments as random integers
17              int x = (int)(Math.random() * 1000) % (XSIZE - 1);
18              int y = (int)(Math.random() * 1000) % (YSIZE - 1);
19              int width = (int)(Math.random() * 1000) % (XSIZE - 1 - x);
20              int height = (int)(Math.random() * 1000) % (YSIZE - 1 - y);
21              int red = (int)(Math.random() * 1000) % (256);
22              int green = (int)(Math.random() * 1000) % (256);
23              int blue = (int)(Math.random() * 1000) % (256);
24
25              // set random color
26              g.setColor(new Color(red, green, blue));
27
28              // draw filled rectangle: random position, random proportions
29              g.fillRect(x, y, width, height);
30
31              temp = t2;
32           }
33        }
34     }
35     static final int XSIZE = 250;
36     static final int YSIZE = 150;
37  }
```

Figure 3. `RandomQuilt.java`

A screen snapshot of this applet after about 10 seconds of execution is shown in Figure 4. Of course, it looks much better in color. Note that the pattern of patches continually changes, with a new patch appearing every 100 ms (ten times per second).

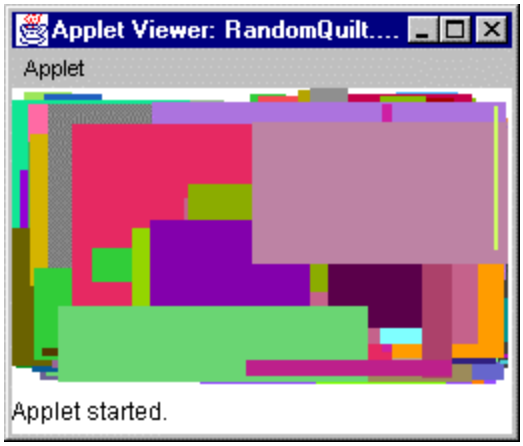


Figure 4. Output of `RandomQuilt` applet

The program executes in an infinite loop and must be terminated by entering Control-c on the keyboard from the DOS window, or by closing the applet window. The `currentTimeMillis()` method of the `System` class is used in the same manner as in the `CountSeconds` and `RandBits` programs discussed earlier.

Within the main loop, seven random numbers are generated for each new rectangle (lines 17-23). Four numbers specify the *x-y* location and *width* and *height* of the rectangle, and three numbers specify the *red*, *green*, and *blue* components of the rectangle's color. Since the `random()` method returns a `double` between 0.0 and 1.0, the result is multiplied by 1000, then cast to an `int`. The result is a random integer between 0 and 1000. This is scaled to an appropriate range using the mod (`%`) operator. The "appropriate range" in the case of the rectangle parameters is such that the rectangle fully lies within the graphics window. The appropriate range for the color parameters is 0-255.

With these seven random numbers, the drawing color is set in line 26 and a filled rectangle is drawn in line 29. The `setColor()` and `fillRect()` methods are in the `Graphics` class in the `java.awt` package (see the Java API documentation).

Date and Time Classes

Did you know that the last minute of 1995 had 61 seconds? This fact may have eluded the creators of *Trivial Pursuit*, but it did not pass unnoticed to the creators of Java. If you don't really care about things like this, you may find Java's date and time support frustratingly complicated. However, if you desire software that supports international interpretations of date and time, you've come to the right programming language. Whether your interest lies in time zones, daylight savings, a lunar calendar for your favourite corner of planet Earth, or micro-adjustments for discrepancies in the earth's rotation, there is support in Java for creating accurate calendars and clocks to meet your needs.

Dates and times in Java are based on coordinated universal time (UTC). This is similar to Greenwich mean time (GMT), with an additional correction to compensate for the earth's non-uniform rotation. In the UTC time system, leap seconds are occasionally added at the end of a year, hence the 61 seconds in the last minute of 1995.

We will not attempt a thorough discussion of Java's date and time classes. For more details consult the Java API documentation. Table 1 summarizes the most important classes for creating and working with date and time objects.

Table 1. Date and Time Classes

Class (package)	Summary
`Calendar` `(java.util)`	an abstract base class for converting between a `Date` object and a set of integer fields such as YEAR, MONTH, DAY, HOUR, and so on. Subclasses of `Calendar` interpret a `Date` according to the rules of a specific calendar system. At present, there is one concrete subclass of `Calendar`: `GregorianCalendar`
`Date` `(java.util)`	a class that represents a specific instant in time, with millisecond precision
`DateFormat` `(java.text)`	an abstract class for date/time formatting subclasses which formats and parses dates or time in a language-independent manner. At present, there is one concreate subclass of `DateFormat`: `SimpleDateFormat`
`GregorianCalendar` `(java.util)`	a concrete subclass of `Calendar` that provides a Gregorian calendar
`SimpleDateFormat` `(java.text)`	a concrete subclass of `DateFormat` for formatting and parsing dates in a locale-sensitive manner. It allows for formatting (date-to-text), parsing (text-to-date), and normalizing
`SimpleTimeZone` `(java.util)`	a concrete subclass of `TimeZone` that represents a time zone for use with a Gregorian calendar. This class does not handle historical changes.
`TimeZone` `(java.util)`	a class that represents a time zone offset, and also figures out daylight savings time
`Locale` `(java.util)`	a class that represents a specific geographical, political, or cultural region.

Let's examine a few simple demo programs that exercise Java's date and time classes.

Date and Time Formats

The `DemoDateTime` program uses the `Date` class and the `SimpleDateFormat` class to obtain the current date and time and print it in some common formats (see Figure 1).

```
 1  import java.util.*;
 2  import java.text.*;
 3
 4  public class DemoDateTime
 5  {
 6     public static void main(String[] args)
 7     {
 8        // get current date and time
 9        Date currentTime = new Date();
10
11        // create SimpleDateFormat object with default format
12        SimpleDateFormat sdf = new SimpleDateFormat();
13
14        // print current date and time using default format
15        System.out.println(sdf.format(currentTime));
16
17        // set a new format
18        sdf.applyPattern("'Date:' MMMM d, yyyy  'Time:' hh:mm:ss a zzz");
19
20        // print again using new format
21        System.out.println(sdf.format(currentTime));
22
23        // set a format for day of week
24        sdf.applyPattern("'Today is' EEEE");
25
26        // print day of the week using new format pattern
27        System.out.println(sdf.format(currentTime));
28     }
29  }
```

Figure 1. `DemoDateTime.java`

This program generated the following output:

```
1/2/99 5:15 AM
Date: January 2, 1999  Time: 05:15:01 AM EST
Today is Saturday
```

This was a correct representation of the date and time on the local computer when the program was run. The program begins by instantiating a `Date` object (line 9) named `currentTime` and a `SimpleDateFormat` object (line 12) named `sdf`. The default constructor (no arguments) is used in both cases. The current date and time is printed in line 15 using the default format, and this appears in the first line of output above. Within the print statement, the following expression appears:

```
sdf.format(currentTime)
```

This method returns a `String` representation of the `currentTime` object, formatted according to the `sdf` object.

Let's digress briefly to show how *inheritance* is at work in the expression above. If you look in the Java API documentation for the `format()` method in the `SimpleDateFormat` class, you'll discover that it does not appear in the form above — with one `Date` object as an argument. The `format()` method, as used above, appears in the `DateFormat` class, which is the

superclass to `SimpleDateFormat` class. Since `SimpleDateFormat` is a subclass to `DateFormat`, it inherits all the methods of `DateFormat`, such as `format(String s)`. We will talk about inheritance in more detail later.

If we want a different format for the date and/or time, then the `applyPattern()` method of the `SimpleDateFormat` class is used. This is illustrated in line 18 of Figure 1. The effect of the new format is shown in line 21 by printing the date and time again. This is shown in the second line of output. The characters in the format pattern in line 18 are part of a flexible set of formatting symbols. The complete set is shown in Table 2.

Table 2. Date and Time Formatting Symbols

Symbol	Meaning	Example
G	era designator	AD
y	year	1996
M	month in year	July & 07
d	day in month	10
h	hour in am/pm (1~12)	12
H	hour in day (0~23)	0
m	minute in hour	30
s	second in minute	55
S	millisecond	978
E	day in week	Tuesday
D	day in year	189
F	day of week in month	2 (2nd Wed in July)
w	week in year	27
W	week in month	2
a	am/pm marker	PM
k	hour in day (1~24)	24
K	hour in am/pm (0~11)	0
z	time zone	Pacific Standard Time
'	escape for text	
''	single quote	'

Clearly, the symbols in Table 2 offer many possibilities for working with dates and time. The count of the formatting symbols determines the format. For text fields, if four or more symbols appear, the full form is used (e.g., "`EEEE`" might yield `Tuesday`), otherwise an abbreviated form is used (e.g., "`EEE`" might yield `Tue`). For numeric fields, the number of characters determines the minimum number of digits, with shorter numbers padded left with zeros. The year symbol is handled differently: If the count is 2, year is truncated to 2 digits. The month-in-year field also has a special interpretation. If three or more characters appear, the text form is used (e.g., "`MMMM`" might yield `January`), otherwise the numeric form is used (e.g., "`MM`" might yield `01`). Punctuation characters in the formatting string are presented as such. Additional alpha characters are included by enclosing them in single quotes.

The formatting pattern is changed in line 24 to specify only the day of the week, as printed in line 27 and shown in the third line of output.

Real-Time Clock

Time and tide wait for no man (Chaucer, c. 1390) — and neither does the host system's clock. The `DemoDateTime` program illustrated flexible ways to control the printed format of dates and times. However, the three lines of output all represented the single instance in time when the `Date()` constructor was called (see line 9 in Figure 1). In this section, we'll examine a real-time clock that outputs a new date and time each second.

To update the time "each second", we need access to the "seconds" field in the current date and time. Although we could examine the strings created by the `format()` method of the `DateFormat` class, this is awkward. A better approach is to use a calendar object. Objects of Java's `Calendar` class (and its subclass, `GregorianCalendar`) can be updated using the `setTime()` method so that they reflect the current time. Furthermore, individual fields (seconds, day of the week, etc.) can be accessed using the `set()` and `get()` methods. Let's illustrate this in a demo program. The program `DemoClock` creates a real-time clock on the host system's display (see Figure 2).

```
 1  import java.util.*;
 2
 3  public class DemoClock
 4  {
 5     public static void main(String[] args)
 6     {
 7        Calendar today = new GregorianCalendar();
 8        int seconds = today.get(Calendar.SECOND);
 9        while (true)
10        {
11           today.setTime(new Date());
12           int newSeconds = today.get(Calendar.SECOND);
13           if (newSeconds != seconds)
14           {
15                System.out.print(today.getTime() + "\r");
16                seconds = newSeconds;
17           }
18        }
19     }
20  }
```

Figure 2. `DemoClock.java`

This program generated the following output when it was executed on the local system:

```
Sat Jan 02 10:34:28 EST 1999
```

The output was updated every second. By using the `print()` method, instead of `println()`, and by appending "`\r`" to the string representing the current time, each update overwrites the previous update (see line 15).[1] This creates a nice visual effect for the output.

[1] Recall that "`\r`" represents "return", whereas "`\n`" represents "newline". The effect of printing "`\r`" is to return the pointer to the left side of the current line without advancing to the next line. Characters subsequently written overwrite previous characters.

A new `Calendar` object is instantiated in line 7 and a reference to it is assigned to the object variable `today`. Note that the constructor for a `GregorianCalendar` object is used. Since `GregorianCalendar` is a subclass of `Calendar`, this is a legitimate operation. The instantiated object is a `Calendar` object bearing the characteristics of the current date and time as per the Gregorian calendar — the calendar used in most of the world.

Initially, `today` holds a representation of the current date and time. In line 8, the seconds value is extracted from `today`, and assigned to the `int` variable `seconds`. This occurs using the `get()` method of the `Calendar` class with the field `Calendar.SECONDS` as an argument.

Lines 9-18 contain an infinite loop that outputs the current date and time once per second. (The program is terminated by entering Control-c on the keyboard.) Within the loop, the first task is to update `today` to the current date and time. This happens in line 11 using the `setTime()` method. The argument to `setTime()` is a `Date` object. The expression "`new Date()`" is the argument, and this achieves the desired effect of getting the "current" date and time. Then, the seconds value is extracted from `now` and assigned to the `int` variable `newSeconds` in line 12. The first time line 12 executes, `newSeconds` is likely assigned the same value as `seconds` in line 8. (After all, today's computers are pretty fast!) `newSeconds` and `seconds` are compared in the `if` statement in line 13. If they differ, lines 15-16 execute: The current date and time are printed (line 15) and `newSeconds` is updated to `seconds` (line 16). If they are the same, another iteration of the `while` loop begins and `newSeconds` is updated again.

If the description above sounds familiar, it should. The exact same technique was employed in the `CountSeconds` program presented earlier.

Calendar Entries

Calendar objects are also used to represent important dates in the future, such as anniversaries, holidays, or appointments. A variety of manipulations may be required for such objects. We may want to know how many shopping days are left until Christmas or some other gift-sharing holiday, for example. Or, perhaps we want a reminder when someone's birthday arrives. The `Calendar` and `GregorianCalendar` classes include the methods necessary to design programs for such purposes. Let's look at an example.

In the northern hemisphere, summer usually begins on June 21. The program `DaysToSummer` provides some information that might be of interest to sun seekers (see Figure 3).

```
 1  import java.util.*;
 2
 3  public class DaysToSummer
 4  {
 5     public static void main(String[] args)
 6     {
 7        Calendar today = new GregorianCalendar();
 8
 9        Calendar summerStart = new
10           GregorianCalendar(today.get(Calendar.YEAR), Calendar.JUNE, 21);
11
12        int days = summerStart.get(Calendar.DAY_OF_YEAR) -
13              today.get(Calendar.DAY_OF_YEAR);
14
15        if (days > 0)
16           System.out.println(days + " day(s) to summer");
17        else if (days == 0)
18           System.out.println("Summer begins today!");
19        else
20           System.out.println("Summer began " + -days + " day(s) ago");
21     }
22  }
```

Figure 3. `DaysToSummer.java`

There are three possible outputs from this program. If it is executed between January 1 and June 20 in any year, the output is

n `days to summer`

where *n* is the number of days to the beginning of summer. If it is executed on June 21, the output is

`Summer begins today!`

If it is executed after June 21, the output is

`Summer began` *n* `day(s) ago`

where *n* is the number of days since summer started.

The program begins by creating two `Calendar` objects. The first is `today`, representing the current day (line 7), and the second is `summerStart` representing June 21 in the current year (lines 10-11). Both objects bear the characteristics of an instance in time according to the rules of the Gregorian calendar. Two different constructors are used. If the default constructor (no arguments) is used, the instantiated object represents the current date and time. The constructor in line 10 uses three arguments. The first is the expression `today.get(Calendar.YEAR)`, representing the current year. The second is `Calendar.JUNE`, which is a public data field of the `Calendar` class. It happens to be an `int` data constant. The specific value is irrelevant for our purposes. The third argument is 21, an integer representing the day of the month when summer begins.

The number of days until the beginning of summer is calculated in lines 12-13 using two calls to the `get()` method, with `Calendar.DAY_OF_YEAR` as an argument. These return integers equal to the "day number in the year" of the `Calendar` objects specified: `summerStart` and `today`. They are subtracted and assigned to the `int` variable `days`. There are three possibilities for `days`. It is positive if summer has yet to arrive. It is negative if summer has

already started. It is zero if today is the first day of summer. These three possibilities are tested for in lines 15-17, and an appropriate message is printed.

Class Hierarchy - Date and Time Classes

Figure 4 illustrates the Java's date and time class in a class hierarchy diagram. The `TimeZone`, `SimpleTimeZone`, and `Locale` classes were not discussed in this section. (You may want to investigate these on your own.)

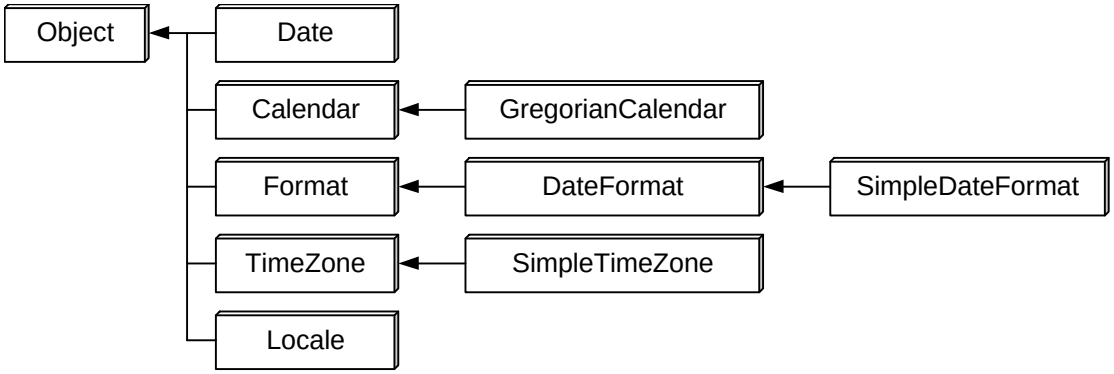


Figure 4. Class hierarchy - date and time classes

Chapter 5

Defining Methods

Defining Methods – Why?

In the preceding sections, we put Java to work in solving interesting problems. The demo programs grew to a healthy size, however. The `CountWords2.java` program, as an example, took about 40 lines of code and had six levels of indentation at the deepest spot. Coping with this increase in complexity is the topic of the next several sections. We'll learn how to define custom *methods* to subdivide complex programming tasks into smaller tasks that are easy to understand and easy to manage.

A method is a *function*. The term "function" has a heritage in computer science dating back several decades; however the term is not used in Java. If you have programming experience with a language like C or Pascal, and you are just learning Java, you may be more comfortable with the term "function". If you think "function" when you read "method", it may help ease you into the world of Java. Make it a personal goal, however, to wean yourself of the term "function". Once you begin to think in terms of methods — constructors, class methods, and instance methods — you are well underway to becoming a Java programmer.

All the methods presented in this chapter are examples of *class methods*, as opposed to *instance methods*. The distinction between the two was elaborated in a preceding section, but will re-surface here. We will learn to defined instance methods later when we learn to define classes.

Any programming problem that can be solved using methods, can be solved without methods. So, what's the big deal? Are methods just another of those tools, like the `break` statement or the selection operator, that exist for our convenience but aren't really necessary? Not at all. Methods are an essential Java programming tool. It would not be possible in practice to solve "large" programming problems without methods. The task would grow in complexity until it was impossible for a mere mortal to understand how all the parts worked together.

There are at least three reasons why we use methods: (i) to allow us to cope with complex problems, (ii) to hide low-level details that otherwise obscure and confuse, and (iii) to reuse portions of code. Let's examine each of these points in turn.

Complexity

As programs gain in complexity, they generally gain in size, and, sooner or later, they become unruly. When confronting hundreds of lines of code, it is easy to confuse how pieces fit together in achieving an overall goal. One of most powerful techniques to cope with complexity in computer programs is affectionately known as "divide and conquer" — that is, to take a complex task and divide it into smaller, more easily understood tasks. In Java, those "smaller, more easily understood tasks" are implemented as *methods*. Once a method is defined, it is "put aside" and used. The task of the method is thereafter "solved", and we needn't concern ourselves with the details any longer.

Hiding Details

Another important use of methods is to create "black boxes". Once a method is defined, it can be used in a program. At the level of using it, we needn't concern ourselves with how the method's task is performed. The details are hidden, and that's just great because our thoughts are focused "higher up" — in solving a big problem that exploits the solutions to small problems in achieving a goal. This is analogous to management hierarchies in corporations: Senior managers delegate tasks to junior managers because solving those tasks themselves is distracting. When we use a method, we are "delegating" the task of solving a low-level problem to a method. To solve it

oneself is distracting. We are, in a sense, treating the method as a black box, because we accept the result without concern for the details. This is illustrated in Figure 1.

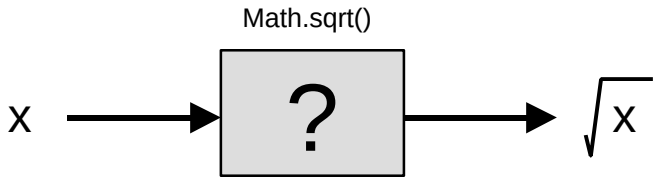


Figure 1. A method as a black box

The question mark in Figure 1 emphasizes that the details of calculating a square root are hidden from us. Delegating tasks to methods is a great luxury, and we want to exploit it to the fullest. The `Math.sqrt()` method is an excellent example because it is a common task that we call on frequently in solving high-level problems, such as finding the length of the hypotenuse (z) of a right-angle triangle:

$$z = \sqrt{x^2 + y^2}$$

The low-level details of calculating a square root are hidden, and that's fine because we want to focus our efforts higher up, in solving a problem that just happens to use a square root operation as part of the solution. We treat the `Math.sqrt()` method as a black box.

Reuse

Once a task is packaged in a method, that method is available to be accessed, or "called", from anywhere in a program. The method can be reused. It can be called more than once in a program, and it can be called from other programs. This is illustrated in Figure 2.

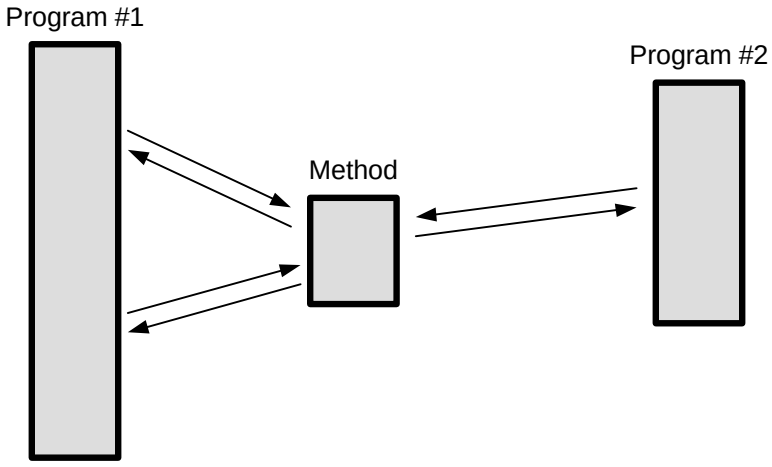


Figure 2. Method reuse

Reuse is an important concept in object-oriented programming languages like Java. The practice is sometimes called, "Write once, use many". Rather than "reinventing the wheel", we build on previous efforts or on the efforts of other programmers — we reuse what precedes us.

In the next section we'll learn how to define simple methods in the Java language.

Method Syntax

Let's begin with a simple demo program that uses a method that we define. The operation is deliberately trivial to allow us to focus on the syntax. The goal is to learn how methods are defined and used. The program `DemoMethod` defines and uses a method that computes the larger of two numbers (see Figure 1).

```
 1  import java.io.*;
 2
 3  public class DemoMethod
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        BufferedReader stdin =
 8           new BufferedReader(new InputStreamReader(System.in), 1);
 9
10        System.out.print("Enter an integer: ");
11        int x = Integer.parseInt(stdin.readLine());
12
13        System.out.print("Enter another one: ");
14        int y = Integer.parseInt(stdin.readLine());
15
16        int z = DemoMethod.largerOf(x, y);
17        System.out.println("The larger one is " + z);
18     }
19
20     public static int largerOf(int a, int b)
21     {
22        if (a > b)
23           return a;
24        else
25           return b;
26     }
27  }
```

Figure 1. `DemoMethod.java`

A sample dialogue follows:

```
PROMPT>java DemoMethod
Enter an integer: 35
Enter another one: 24
The larger one is 35
```

Obviously, this program could be written without the `largerOf()` method: It's pretty simple. However, `DemoMethod` serves our goal of introducing the syntax for defining and using methods. The very simple `largerOf()` method compares two integers and returns the larger of the two. The method is defined in lines 20-26; it is used in line 16.

The syntax in line 16 is similar to our use of pre-defined methods in previous programs, such as the `Math.sqrt()` method. The `sqrt()` method is a class method of the `Math` class. The `largerOf()` method is a class method of the `DemoFunction` class. Since the `Math` class and its methods are defined outside the program where `sqrt()`is used, the prefix "`Math.`" is

required to inform the compiler of the location of the method.[1] However, the `largerOf()` method is defined in the class `DemoMethod`, so in line 16, the prefix "`DemoMethod.`" is unnecessary. The prefix is used simply to emphasize that `largerOf()` is a class method, just like `sqrt()`. The usual syntax for line 16 is

```
int x = largerOf(x, y);
```

Method Signature

The `largerOf()` method is defined in lines 20-26. The first line of the method definition is

```
public static int largerOf(int a, int b)
```

This is the method's *signature*. It contains all the information needed to use the method. The documentation for each method in Java's API contains only the method's signature and a brief description of its operation. As programmers using the method, that's all we need to know. Let's walk through the signature for the `largerOf()` method.

The reserved word "`public`" is a *modifier*; it specifies the *visibility* attribute of the method. Most Java methods are *public*, meaning they are accessible from other methods, perhaps from methods in other classes. Visibility attributes are discussed in more detail later.

The reserved word "`static`" is also a modifier; it identifies the method as a *class method* (sometimes called a *static method*). It means that the method is not called through an instance of a class but, rather, through the class itself. Understanding this is tricky, so let's review the syntax for calling a class method vs. calling an instance method. (This was discussed earlier.) The method `sqrt()` of the `Math` class is a "class method". It might be called as follows:

```
double x = 25.0
double y = Math.sqrt(x);  // sqrt() is a 'class method'
```

The method `substring()` of the `String` class is an "instance method". It might be called as follows:

```
String s1 = "hello";
String s2 = s1.substring(0, 1);  // substring() is an 'instance method'
```

Do you see the difference? Preceding `sqrt()` is "`Math.`" which identifies the class in which `sqrt()` is defined. Preceding `substring()` is "`s1.`" which is an instance variable of the `String` class. "`Math`" is the name of a class (hence "class method"), whereas "`s1`" is the name of an instance variable (hence "instance method").

The preceding discussion is put in terms of the `largerOf()` method as follows: the `largerOf()` method is a static method, or a class method, of the `DemoFunction` class.

Returning to the signature for the `largerOf()` method, the next word is the reserved word "`int`". This identifies the return type of the method. The `largerOf()` method returns a value of type `int`. Anywhere an integer value can be used — for example, in an expression — the `largerOf()` method can be used. All Java methods must include a return type in their

---

[1] The `Math` class is part of the `java.lang` package. It is usually necessary to further identify the location of methods and classes with an `import` statement at the beginning of a program, for example "`import java.lang.*;`". Since the `java.lang` package contains the basic classes required for all Java programs, its location is known by the compiler by default.

definition. In some cases, a method has nothing to return, (e.g., `println()`), and is defined to return a `void`.

Next comes the name of the method followed by a set of parentheses. The name of the method conforms to the naming conventions for Java identifiers given earlier. Within the parentheses are the arguments passed to the method. In the case of `largerOf()`, it receives two arguments of type `int`. The names of the arguments are arbitrary. They are used in the definition of the method (see lines 22-25), but they have no relationship to the names of variables in the calling program.

Immediately following the signature is the definition of the method within a set of braces. The `largerOf()` method is defined in lines 21-26. The operation is pretty simple, so we needn't dwell on the details. The reserved word "`return`" causes an immediate exit from the method, with control returning to the calling method. `return` is followed by an expression, a variable name, or the value to be returned to the calling program. A compile error occurs if the type of the value returned does not match the type appearing the signature (notwithstanding automatic promotion, for example, of `int` to `double`).

Let's explore the definition of class methods by "updating" several of the demo programs from previous chapters.

Count Words - Revisited

`CountWords3` (see Figure 2) is a reworked version of `CountWords2`, except the code to determine if a word contains only letters is taken out of the `main()` method and placed in a dedicated method called `isAlphaWord()`.

```
 1  import java.io.*;
 2  import java.util.*;
 3
 4  public class CountWords3
 5  {
 6     public static void main(String[] args) throws IOException
 7     {
 8        BufferedReader stdin =
 9           new BufferedReader(new InputStreamReader(System.in), 1);
10
11        String line;
12        String word;
13        StringTokenizer words;
14        int count = 0;
15
16        // process lines until no more input
17        while ((line = stdin.readLine()) != null)
18        {
19           // process words in line
20           words = new StringTokenizer(line);
21           while (words.hasMoreTokens())
22           {
23              word = words.nextToken();
24              if (isAlphaWord(word))
25                 count++;
26           }
27        }
28        System.out.println("\n" + count + " alpha words read");
29     }
30
31     public static boolean isAlphaWord(String w)
32     {
33        for (int i = 0; i < w.length(); i++)
34        {
35           String c = w.substring(i, i + 1).toUpperCase();
36           if (c.compareTo("A") < 0 || c.compareTo("Z") > 0)
37              return false;
38        }
39        return true;
40     }
41  }
```

Figure 2. `CountWords3.java`

Have a look at lines 21-26 in Figure 2 and notice how easy they are to interpret. All the "dirty work" is gone. The `isAlphaWord()` method is treated as a black box: It returns a `boolean` which is `true` if the word contains only letters of the alphabet. Since it returns a `boolean`, the method can appear anywhere a `boolean` variable can be used, for example, in a relational expression. In fact, it forms the entire relational expression in the `if` statement that determines whether or not to increment the `int` variable `count` (line 24). Recall, that a relational test is always a test for "`true`", so the expression

```
isAlphaWord(word)
```

is identical to

```
isAlphaWord(word) == true
```

The definition of the `isAlphaWord()` method appears in lines 31-40, and it contains what we fondly referred to earlier as "the dirty work" — in this case, the process of scanning characters in a word to determine if they are all letters. The details were discussed earlier, so we won't say more here.

Find Palindromes - Revisited

The `Palindrome` program presented in Chapter 4 is re-worked in Figure 3 using a method.

```
 1  import java.io.*;
 2  import java.util.*;
 3
 4  public class Palindrome2
 5  {
 6     public static void main(String[] args) throws IOException
 7     {
 8        // open keyboard for input (call it 'stdin')
 9        BufferedReader stdin =
10           new BufferedReader(new InputStreamReader(System.in), 1);
11
12        // prepare to extract words from lines
13        String line;
14        StringTokenizer words;
15        String word;
16
17        // main loop, repeat until no more input
18        while ((line = stdin.readLine()) != null)
19        {
20           // tokenize the line
21           words = new StringTokenizer(line);
22
23           // process each word in the line
24           while (words.hasMoreTokens())
25           {
26              word = words.nextToken();
27              if (word.length() > 2 && isPalindrome(word))
28                 System.out.println(word);
29           }
30        }
31     }
32
33     public static boolean isPalindrome(String w)
34     {
35        int i = 0;
36        int j = w.length() - 1;
37        while (i < j)
38        {
39           if ( !(w.charaAt(i) == w.charAt(j)) )
40              return false;
41           i++;
42           j--;
43        }
44        return true;
45     }
46  }
```

Figure 3. `Palindrome2.java`

Note how easy it is to understand the act of checking if words of three or more characters are palindromes (see line 27). Both "`word.length() > 2`" and "`isPalindrome(word)`" are relational expressions that return `boolean` values. They are connected by the logical AND (&&) operator. If both expressions are `true`, the overall result is `true` and the word is echoed to the standard output in line 28. At the level of using the `isPalindrome()` method, we nee not concern ourselves with the details: We just use the method for its intended purpose. By removing the code to check for palindromes, we have eliminated clutter in the `main()` method that otherwise obscures the overall operation.

The definition of the `isPalindrome()` method appears in lines 33-45. The code is identical to that appearing in `Palindrome.java`, except it is packaged in a class method. The method's signature (line 33) declares that the method receives a `String` argument and returns a `boolean` result.

Reversing Characters in a String

Now let's return to another task we met earlier. The program `StringBackwards2` is the same as `StringBackwards`, except for two small changes. First, the original string is now inputted via the keyboard, and, second, the code to reverse the characters in the string is packaged in a method (see Figure 4).

```
 1  import java.io.*;
 2
 3  public class StringBackwards2
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        BufferedReader stdin =
 8           new BufferedReader(new InputStreamReader(System.in), 1);
 9
10        System.out.print("Enter a message string: ");
11        String s = stdin.readLine();
12        s = backwards(s);
13        System.out.print(s);
14     }
15
16     public static String backwards(String s)
17     {
18        String b = "";
19        for (int i = 0; i < s.length(); i++)
20           b = s.substring(i, i + 1) + b;
21        return b;
22     }
23  }
```

Figure 4. `StringBackwards2.java`

A sample dialogue follows: (User input is underlined.)

```
PROMPT>java StringBackwards2
Enter a message string: Java is fun!
!nuf si avaJ
```

Once again, we see a clear improvement in readability in the main program (lines 11-13) by taking a "divide-and-conquer" approach. The code to reverse the characters in a string is

packaged in the `backwards()` method. The method is defined in lines 16-22; it is used in line 12.

Note that the definition of the `backwards()` method uses an identifier, `s`, which is the same as an identifier in the main program (lines 11, 12, & 13). The two are not related. Identifiers used in the definition of methods have no relationship to identifiers of the same name in the calling program or in the definition of other methods. Have a second look at Figure 4 and make sure you are comfortable with this point. The same identifiers were used in `main()` and in `backwards()` in Figure 4 specifically to bring out this point. If the three uses of `s` in the definition of `backwards()` (lines 16, 19, & 20) are changed to, for example, `originalString`, the effect is the same. We will say more about this later in our discussion of variable scope.

We'll return to `StringBackwards2` later in our discussion of recursion and debugging.

Formatted Output

Now that we are comfortable with defining methods, we can think about packaging useful operations into methods and working them into Java programs. In this section, we'll examine some custom methods to assist in generating formatted output.

The standard library for C programming language includes a function known as `printf()`, for "print formatted". `printf()` receives a series of variables and a format string as arguments. The format string uses a custom syntax to specify how each variable should appear (i.e., the field width, the alignment within the field, and the type of padding). Unfortunately, an equivalent method does not exist in the Java API. That's OK, because we can "roll our own". Although we'll not attempt a full implementation of a `printf`-equivalent method, we can craft a few simple methods to help generate nicely formatted output.

As a first cut, let's examine how to control the format for integers printed on the standard output. The program `TableOfCubes` outputs a table of cubes for the integers 1 through 10 (see Figure 1).

```
 1  import java.io.*;
 2
 3  public class TableOfCubes
 4  {
 5     public static void main(String[] args)
 6     {
 7        System.out.print("=============\n"
 8                       +  " x  Cube of x\n"
 9                       +  "--- ---------\n");
10        int x = 0;
11        String s1;
12        String s2;
13        while (x <= 10)
14        {
15           s1 = formatInt(x, 3);
16           s2 = formatInt(x * x * x, 10);
17           System.out.println(s1 + s2);
18           x++;
19        }
20        System.out.println("=============\n");
21     }
22
23     // format int 'i' as a string of length 'len' (pad left with spaces)
24     public static String formatInt(int i, int len)
25     {
26        String s = "" + i;
27        while (s.length() < len)
28           s = " " + s;
29        return s;
30     }
31  }
```

Figure 1. `TableOfCubes.java`

This program generates the following output:

```
=============
 x  Cube of x
--- ---------
  0         0
  1         1
  2         8
  3        27
  4        64
  5       125
  6       216
  7       343
  8       512
  9       729
 10      1000
=============
```

The output contains two nicely-formatted columns of integers. The method `formatInt()` helps. It is defined in lines 24-30 and is used twice, in lines 15 and 16.

The entire method definition contains only seven lines. As seen in the signature, `formatInt()` receives two integer arguments and returns a reference to a new `String` object. The two arguments are the integer to be formatted, and the minimum length of the string. The first task is to convert the integer to a string. This is performed easily by concatenating the integer with an empty (line 26). Then, a two-line `while` loop pads the string on the left with spaces until the desired length is achieved (lines 27-28). The result is returned to the calling program in line 29.

In the main program `formatInt()` is called twice. In line 15, the arguments `x` and 3 are passed and the `String` reference returned is assigned to `s1`. In line 16, the arguments `x * x * x` and 10 are passed and the `String` reference returned is assigned to `s2`. The two strings are concatenated in the print statement in line 17. The string printed is 13 characters wide, with the original integer right-aligned in the first three positions, and the cube of the integer right-aligned in the next 10 positions.

The situation is more awkward for floating point numbers, since we must deal with padding left with spaces and with truncating decimal places or padding right with zeros. The program `TableOfSquareRoots` illustrates one approach to generating formatted output of floating point numbers (see Figure 2).

```
 1  public class TableOfSquareRoots
 2  {
 3     public static void main(String[] args)
 4     {
 5        System.out.print("====================\n"
 6                       +  " x  Square Root of x\n"
 7                       +  "--- ----------------\n");
 8
 9        for (int x = 0; x <= 200; x += 25)
10        {
11           String s1 = formatInt(x, 3);
12           String s2 = formatDouble(Math.sqrt(x), 17, 4);
13           System.out.println(s1 + s2);
14        }
15        System.out.println("====================\n");
16     }
17
18     // format int 'i' as a string of length 'len' (pad left with spaces)
19     public static String formatInt(int i, int len)
20     {
21        String s = "" + i;
22        while (s.length() < len)
23           s = " " + s;
24        return s;
25     }
26
27     // format double 'd' as a string of length 'len' & 'dp' decimal places
28     public static String formatDouble(double d, int len, int dp)
29     {
30        String fx = (dp <= 0) ? "" + Math.round(d) : "" + trim(d, dp);
31        while (fx.substring(fx.indexOf(".") + 1, fx.length()).length() < dp)
32           fx += "0";
33        while (fx.length() < len)
34           fx = " " + fx;
35        return fx;
36     }
37
38     // trim double 'd' to have 'dp' decimal places
39     public static double trim(double d, int dp)
40     {
41        double factor = Math.pow(10.0, dp);
42        return (int)(d * factor) / factor;
43     }
44  }
```

Figure 2. `TableOfSquareRoots.java`

This program generates the following output:

```
====================
 x  Square Root of x
--- ----------------
  0           0.0000
 25           5.0000
 50           7.0710
 75           8.6602
100          10.0000
125          11.1803
150          12.2474
175          13.2287
200          14.1421
====================
```

Within 44 lines of code, `TableOfSquareRoots` contains a `main()` method and three custom methods: `formatInt()`, `formatDouble()`, and `trim()`. The left-hand column of output displays the integers from 0 to 200 in increments of 25. Each integer appears right-justified in a three-character column. The `formatInt()` method does the work, as in the previous example. The right-hand column displays the square roots of the integers. Each root contains four decimal places of precision and appears right-justified in a 17-character column. The formatted strings for the integers and the square roots of the integers are generated in the `main()` method in lines 11 and 12, respectively.

Note in the call to the `formatDouble()` method in line 12 that three arguments are passed. The first is the `double` variable to format into a string. The expression `Math.sqrt(x)` appears; so, the value to format is the square root of the integer. The second argument is 17, which is the minimum length of the string, and the third argument is 4, which is the number of decimal places of precision.

The definition of `formatDouble()`appears in lines 28-36. The string that is eventually returned, `fx`, is declared and initialized in line 30. The assignment uses a selection operator, expanded as follows:

```
if (dp <= 0)
   fx = "" + Math.round(d);
else
   fx = "" + trim(d, dp);
```

So, if the method is called specifying less than one decimal place of precision, `fx` is initialized with a string representation of the integer returned by `Math.round(d)`. This is the closest integer to the double argument `d`. If the method is called specifying one or more decimal places of precision, `fx` is initialized with a string representation of the value returned by `trim(d, dp)`. The `trim()` method returns a `double` equal to `d` with at most `dp` decimal places (see lines 39-43). Lines 31-32 pad `fx` to the right with zeros in the event fewer than `dp` decimal places are present. Lines 33-34 pad `fx` to the left with spaces to fill the field width as specified in the second argument. The final value of `fx` is returned in line 35.

There are a number of deficiencies in the way formatting is performed in `TableOfSquareRoots`. The use of the `pow()` method to calculate a trimming factor in line 41 is overkill, and the technique for timing in line 42 truncates rather than rounds. We'll leave it to you to explore ways to improve these methods.

Input Validation

When humans interact with technology, a great diversity in experience is brought to bear on the task, and things often go amiss. If the technology is well-designed, mistakes or unusual or unexpected interactions are anticipated and accommodated in a safe and non-destructive manner. Computer software is no exception. When a user enters a syntactically incorrect value or selects a disabled feature, for example, the software must recognize, accommodate, and recover from the anomalous condition with minimal disruption. In other words, our software must expect the unexpected. In this section, we will illustrate how to write simple programs that recognize and recover from invalid user input. We call this *input validation*.

Many of our programs were designed to receive input from the keyboard using the `readLine()` method of the `BufferedReader` class. The `readLine()` method returns a string holding a full line of input. If the user entered an integer or floating point number, the string was converted to an `int` using the `Integer.parseInt()` method or to a `double` using the `Double.parseDouble()` method. For example

```
String s = readLine();
int value = Integer.parseInt(s);
```

But, what if the user entered something like "`thirteen`", instead of "`13`"? Here's the reaction from the `DemoMethod` program presented earlier:

```
PROMPT>java DemoMethod
Enter an integer: thirteen
Exception in thread "main" java.lang.NumberFormatException: thirteen
   at java.lang.Integer.parseInt(Compiled Code)
   at java.lang.Integer.parseInt(Integer.java:458)
   at DemoMethod.main(DemoMethod.java:21)
PROMPT>
```

Obviously, thirteen is not our lucky number. The output is hardly what we'd call a user-friendly error message. The program has crashed, and we're left high and dry. The critical term above is "`NumberFormatException`". There was an attempt to parse the string "`thirteen`" into an `int` and the `parseInt()` method didn't like what it found. The string "thirteen" has no correspondence to an integer — at least, not in the eyes of the `parseInt()` method. It reacted by throwing a "number format exception".

In this section, we'll examine a simple way to implement input validation. If the user input is not correctly format, we want to gracefully recover, if possible. And we definitely don't want our program to crash. Input validation is well-suited to implementation in methods; so, this a good place to introduce the topic. We'll re-visit the topic in more detail later when we examine exceptions.

The trick to input validation, as presented in this section, is to inspect the input string *before* passing it to a parsing method. If an invalid string is parsed, then it's too late for graceful recovery. We want to identify the problem before parsing, and recover in a manner that seems appropriate for the given program. The demo program `InputInteger` illustrates one approach to the problem (see Figure 1).

```
 1  import java.io.*;
 2
 3  public class InputInteger
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        // setup 'stdin' as handle for keyboard input
 8        BufferedReader stdin =
 9           new BufferedReader(new InputStreamReader(System.in), 1);
10
11        // send prompt, get input and check if valid
12        String s;
13        do
14        {
15           System.out.print("Enter an integer: ");
16           s = stdin.readLine();
17        }
18        while (!isValidInteger(s));
19
20        // convert string to integer (safely!)
21        int i = Integer.parseInt(s);
22
23        // done!
24        System.out.println("You entered " + i +  " (Thanks!)");
25     }
26
27     // check if string contains a valid integer
28     public static boolean isValidInteger(String str)
29     {
30        // return 'false' if empty string
31        if (str.length() == 0)
32           return false;
33
34        // skip over minus sign, if present
35        int i;
36        if (str.indexOf('-') == 0)
37           i = 1;
38        else
39           i = 0;
40
41        // ensure all characters are digits
42        while (i < str.length())
43        {
44           if (!Character.isDigit(str.charAt(i)))
45              break;
46           i++;
47        }
48
49        // if reached the end of the line, all characters are OK!
50        if (i == str.length())
51           return true;
52        else
53           return false;
54     }
55  }
```

Figure 1. `InputInteger.java`

A sample dialogue with this program follows:

```
PROMPT>java InputInteger
Enter an integer:                    (blank line)
Enter an integer: -                  (only a minus sign entered)
Enter an integer: ninety nine        (alpha characters not allowed)
Enter an integer: -ninety nine       (sorry! try again)
Enter an integer:    99              (leading spaces not allowed)
Enter an integer: 99 98 97           (no spaces allowed)
Enter an integer: 99                 (finally!)
You entered 99 (Thanks!)
```

A variety of incorrect responses are shown above. In all cases, the program reacted by re-issuing the prompt and inputting another line. As evident in the 55 lines of source code, even this simple implementation of input validation is tricky. The good news is that the input routine is packaged in a method. At the level of "using" the method, the code is very clean (see lines 11-18).

Let's look inside the definition of the `isValidInteger()` method. For the purpose of this program, an integer string has the following characteristics: (a) it must contain no leading or trailing spaces, (b) it may contain an optional minus sign at the beginning, and (c) it must contain only digit characters until the end of the string.

The first "processing" task is to ensure that the string is not an empty string. If the line length is zero, the method returns `false` — the string does not contain a valid integer! (See lines 30-32). The next order of business is to check if the first character is a minus sign. An index variable `i` is initialized with 0 if the first character is not a minus sign, or 1 otherwise to skip over the minus sign. The string is scanned from this point to the end checking that each character is a digit. The check is performed in line 44 as follows:

```
if (!Character.isDigit(str.charAt(i)))
```

The character at position `i` in the string is extract using the `charAt()` method of the `String` class and presented to the `isDigit()` method of the `Character` wrapper class. The return value is `true` if the character is a digit, `false` otherwise. If `false` is returned the relational test is `true` — because of the NOT operator (`!`) — and the following statement exectutes. The following statement is a `break`, which causes an immediate exit from the `while` loop. The final value of the index variable `i` is either `str.length()` if all the characters checked out as digits or something less than `str.length()` if the loop terminated early. This condition is check in line 50 and the appropriate boolean result is returned: `true` if the string is a valid integer, `false` otherwise.

Despite the best intentions, the approach to input validation just shown is not as robust as we'd like. If the user enters `12345678999`, for example, the program still crashes with a number format exception. (The largest integer represented by an `int` is $2^{31} - 1 = 2{,}147{,}483{,}647$.) As well, it is not entirely clear that we should reject input with leading or trailing spaces. A far better approach to input validation is to "deal with" the built-in exceptions generated by Java's API classes. We'll learn how to do this later. For the moment, the approach shown in `InputInteger` will serve us well.

Since `isValidInteger()` is a static method, it is available to other programs, provided the `InputInteger.class` file is reachable by the compiler. We could, for example, include the following lines in another Java program:

```
String s;
s = stdin.readLine();
if (InputInteger.isValidInteger(s))
{
   /** process input **/
}
```

The prefix `InputInteger` is simply identifies the the class where the static method `isValidInteger()` is found, much the same as "`Math`" in `Math.sqrt(25)` identifies the class where the `sqrt()` method is found.

Recursion

No discussion of functions or methods is complete without visiting an age-old technique in computer science: *recursion*. A recursive method is a method that calls itself. So, within the definition of the method, there appears a call to the same method. As it turns out, certain computational problems are well-suited to recursion. Any algorithm that computes a result iteratively, where each step builds on a result from the previous step, is a candidate for recursion. We'll illustrate this by developing two example programs.

Factorial of an Integer

A common mathematical operation is to compute the factorial of an integer. If *n* is an integer, then *n*! (read "*n* factorial") is $n \times (n - 1) \times (n - 2)$ ... computed iteratively until the factor 1 appears. So,

```
4! = 4 x 3 x 2 x 1 = 24
```

By definition, 0! = 1. Figure 1 shows the listing for Java program that computes the factorial of an integer. The program includes a non-recursive method named `factorial()`.

```
 1  import java.io.*;
 2
 3  public class Factorial
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        BufferedReader stdin =
 8           new BufferedReader(new InputStreamReader(System.in), 1);
 9
10        System.out.print("Enter a non-negative integer: ");
11        int n = Integer.parseInt(stdin.readLine());
12        if (n < 0)
13        {
14           System.out.println("Oops!");
15           return;
16        }
17        else
18        System.out.println(n + "! = " + factorial(n));
19     }
20
21     public static int factorial(int n)
22     {
23        if (n == 0)
24           return 1;
25        int f = n;
26        for (int i = n - 1; i > 0; i--)
27           f *= i;
28        return f;
29     }
30  }
```

Figure 1. `Factorial` (non recursive solution)

A sample dialogue with this program follows:

```
PROMPT>java Factorial
Enter a non-negative integer: 9
9! = 362880
```

The method `factorial()` is defined in lines 21-29 and is called in line 18. In the method's signature (line 21), we see that it receives an integer argument, `n`, and returns an integer result. Within the method, the first task is to return immediately with "1" if the argument equals "0" (lines 23-24). Otherwise, a local variable `f` is declared and initialized to `n`. Then, a `for` loop is started with a loop control variable, `i`, initialized to `n - 1`. On each pass through the loop, `i` is decremented. Within the loop `f` is reassigned with its existing value multiplied by `i`. When the loop finishes, `f` holds the factorial of the original integer passed to the method. This value is returned in line 28.

Now let's look at a program with the identical behaviour. `Factorial2` is the same as `Factorial` except the method `factorial()`uses recursion to calculate the factorial of an integer (see Figure 2).

```
 1  import java.io.*;
 2
 3  public class Factorial2
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        BufferedReader stdin =
 8           new BufferedReader(new InputStreamReader(System.in), 1);
 9
10        System.out.print("Enter a non-negative integer: ");
11        int n = Integer.parseInt(stdin.readLine());
12        if (n < 0)
13        {
14           System.out.println("Oops!");
15           return;
16        }
17        else
18        System.out.println(n + "! = " + factorial(n));
19     }
20
21     public static int factorial(int n)
22     {
23        if (n == 0)
24           return 1;
25        else
26           return n * factorial(n - 1);
27     }
28  }
```

Figure 2. `Factorial2.java` (recursive solution)

The definition of `factorial()` in lines 21-27 appears much simpler than in the previous example. It's also a lot trickier to understand because of the recursive approach. Note in line 26 that the method calls itself within its own definition. However, each call passes "one less than" the value of the argument passed in the preceding call; so, with each recursive call, we get closer to the correct answer. A written blow-by-blow explanation is not likely to explain as well as a decent visualization, so let's move directly to Figure 3. The figure illustrates the recursive calls to `factorial()` with an original argument of 4.

Step #	Call #	Call value	Return value
1	1	4	
2	2	3	
3	3	2	
4	4	1	
5	5	0	1
6	4		1×1
7	3		$1 \times 1 \times 2$
8	2		$1 \times 1 \times 2 \times 3$
9	1		$1 \times 1 \times 2 \times 3 \times 4$

Figure 3. Visualization of recursive method calls

The left-hand column shows the steps in sequential time order. The second column identifies each call to `factorial()` by number. The method is called a total of five times. The indentation in the second column suggests the level of recursion. The third column identifies the value of the argument passed to the method and the fourth column identifies the value returned by the method. For each call to `factorial()` the value returned should be the factorial of the value passed. And, indeed, this is the case. For each call number in column two, note that the return value (column four) is the factorial of the call value (column three). The specific case for call #1 is highlighted. The call value was 4, and the return value was $1 \times 1 \times 2 \times 3 \times 4 = 24$.

Reversing Characters in a String - Recursion

Let's explore recursion further with an example of a problem met earlier. The `StringBackwards2` program used a method to reverse the pattern of characters in a string. The `StringBackwards3` program performs the same task except using a recursive method (see Figure 4).

```
 1  import java.io.*;
 2
 3  public class StringBackwards3
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        BufferedReader stdin =
 8           new BufferedReader(new InputStreamReader(System.in), 1);
 9
10        System.out.print("Enter a message string: ");
11        String s = stdin.readLine();
12        s = backwards(s);
13        System.out.print(s);
14     }
15
16     public static String backwards(String s)
17     {
18        if (s.length() == 1)
19           return s;
20        else
21        {
22           s = backwards(s.substring(1, s.length())) + s.substring(0, 1);
23           return s;
24        }
25     }
26  }
```

Figure 4. `StringBackwards3.java`

The method `backwards()` is defined in lines 16-25. The method receives a `String` argument and returns a reference to a new `String` object. Within its definition, the method calls itself (line 22). In the recursive call, the argument passed is a substring of the original string. The expression

```
s.substring(1, s.length())
```

represents the original string with the first character removed (see line 22). The first character appears in the expression

```
s.substring(0, 1)
```

which is concatenated to the end of the string returned by the `backward()` method. Thus, the original string is gradually reassembled as the recursion unwinds, with the original characters in reverse order. A visualization of this process is presented in the next section.

The Joy of Recursion

Recursion is a topic you either love or hate. For many students, it's a punishing task to develop a recursive solution to a problem that seems pretty simple without recursion. Like a flowcharting assignment, you may find yourself working backwards — by first solving the problem the way you see it, and then retrofitting your solution to meet the requirements of an assignment. And to this, we have no specific remedy to suggest. However, you'll know you are ready to work with recursion — on your terms — when one day while working on a simple iterative problem, you'll think without warning, "Hey, I can do that using recursion." Our advice: Go for it. You'll have a lot of fun, and when you get the recursive solution working, you'll be sold on it.

There are performance tradeoffs to consider when comparing recursive and non-recursive solutions to programming problems. Non-recursive solutions generally use less memory, since answers are "built up" using one set of variables. Recursive solutions must allocate and maintain a set of local variables for each recursive call. The memory space cannot be freed-up until each method returns, and so, more and more memory is allocated as the recursion deepens. There is an additional overhead of calling a method numerous times. Nevertheless, recursion is a topic that persists. For some interesting recursive problems see Chapters 18 and 19 in Hofstadter's *Metamagical Themas* [1].

Debugging

Finding and correcting problems, or "bugs", in computer programs can be time consuming and frustrating. It would be great if in this short section a step-by-step process could be articulated to direct you in debugging Java programs. Unfortunately, this is not possible. Debugging is tricky business, and it is extremely difficulty to teach. Although there are custom debugging tools available for languages like Java, they are complex and have a steep learning curve. If you are a professional programmer, then the time invested is worth it. But if you program less than, say, two hours a day, you probably lack the time or expertise to operate a debugger.

In this section, we present some tips on debugging. But be forewarned. You will have — and must have — your own style to problem solving. When crunch time arrives, it is *your* approach you'll adopt, regardless of what you read here.

Learn by Experience

If there is one point worth emphasizing, it is this: You cannot learn computer programming by reading a book. You've got to roll-up your sleeves and dig-in. Similarly, you cannot learn debugging by reading about it. You must write programs to learn programming, and you must make and correct mistakes along the way to hone your debugging skills. Every time you fix a compile error or a logic error, you get a little smarter. Take the initiative to write programs to exercise new operators, classes, or methods. Learn by experience. Write programs, and write lots of them.

Stepwise Refinement

Write little programs to do small things to confirm your understanding. If they work, you got it right. If they don't, you missed something. Obviously, the less code introduced at once, the better. The worst possible scenario in debugging is confronting too many problems at once. So, write programs incrementally — one step at a time. This is called *stepwise refinement*. Obtain closure on small sections of code, and test them before proceeding. For example, if you need a custom method, start with the shell and fill in the details later. Let's say you need a method called `complexJob()`. You might organize your program initially as follows:

```
public static void main(String[] args)
{
   ...
   double x = complexJob(a, b);
   System.out.println(x);
   ...
}

public static double complexJob(double arg1, double arg2)
{
   return 0;
}
```

The idea is to get the method's definition and call sequence up and running "structurally", before adding the details. If you have problems in the structure *and* in the details, debugging is harder because too many problems are confronted at once. When you add the details, the same rule applies: Add a few lines of code, get closure on them, and compile and test them before proceeding. Take it once step at a time.

Compile Errors

A compile error is a *syntax* error. You have broken a rule in the Java language, and it must be fixed before proceeding. It is common to get a lot of compile errors from a small typing mistake, like misplacing a semicolon. The most important compile error is the first one generated, so focus your attention on it first. You may be surprised to discover that, having eliminated the first compile error, others vanish as well.

The flip side of the coin is this: Even though you have only one compile error, don't be fooled into thinking your program is almost working. Some errors are so drastic that the compiler "gives up" early on and outputs just one error message. Having fixed this, you may discover dozens of new compile errors. Welcome to debugging!

Run-time Errors

A run-time error is a *semantic* error. The code checked-out fine by the compiler, but when the program executed something ran amuck. There is an error in the meaning, or semantics, of the program's behaviour or in the logical flow of the program. Such errors (e.g., accessing a non-existing array element) are not caught by the compiler because it cannot anticipate changes in variables, and the effect of such, as a program executes.

The symptoms for run-time errors are often strange and seemingly unrelated to the underlying problem in the code. Because of this, run-time errors are generally more difficult to correct than compile errors. This hits home at a point made earlier: Take it one step at a time. If you write a lot of code before testing, you are asking for trouble.

There are two broad categories of run-time errors: those that cause the program to crash, and those that cause the program to generate incorrect results or to behave improperly. Both are semantic errors: the former are errors in how you used Java, the latter are errors in how you approached the problem.

Program Traces

One of the simplest debugging techniques is to add a *program trace* to a program. A program trace is simply a print statement that generates intermediate results. By outputting intermediate results, you can observe a program's behaviour *as it executes*. Let's revisit two programs seen earlier and retrofit them with program traces. The program `Palindrome3` is the same as `Palindrome2`, except program traces are added to output intermediate results (see Figure 1).

```
 1  import java.io.*;
 2  import java.util.*;
 3
 4  public class Palindrome3
 5  {
 6     public static void main(String[] args) throws IOException
 7     {
 8        // open keyboard for input (call it 'stdin')
 9        BufferedReader stdin =
10           new BufferedReader(new InputStreamReader(System.in), 1);
11
12        // prepare to extract words from lines
13        String line;
14        StringTokenizer words;
15        String word;
16
17        // main loop, repeat until no more input
18        while ((line = stdin.readLine()) != null)
19        {
20           // tokenize the line
21           words = new StringTokenizer(line);
22
23           // process each word in the line
24           while (words.hasMoreTokens())
25           {
26              word = words.nextToken();
27              if (word.length() > 2 && isPalindrome(word))
28                 System.out.println(word);
29           }
30        }
31     }
32
33     public static boolean isPalindrome(String w)
34     {
35        int i = 0;
36        int j = w.length();
37        while (i < j)
38        {
39           System.out.println(w.substring(i, i + 1)
40                 + " <---> " + w.substring(j - 1, j));
41           if (!w.substring(i, i + 1).equals(w.substring(j - 1, j)))
42           {
43              System.out.println("NOT A PALINDROME");
44              return false;
45           }
46           i++;
47           j--;
48        }
49        System.out.println("Yes, it's a palindrome");
50        return true;
51     }
52  }
```

Figure 1. `Palindrome3.java`

An example dialogue with this program follows:

```
PROMPT>java Palindrome3
kayak
k <---> k
a <---> a
y <---> y
Yes, it's a palindrome
kayak
momentum
m <---> m
o <---> u
NOT A PALINDROME
^z
```

Two words were inputted: "kayak", which is a palindrome, and "momentum", which is not. Program traces were added in lines 39-40, 43, and 49. The output above is interesting but it is not particularly relevant to the present discussion — because the program works. However, if the program had a run-time error, or generated wrong answers, the output above would shed some light on the problem. For example, the character comparisons in line 41 are shown explicitly in the output, for example "`k <---> k`". If the comparison used the wrong indices, this would surface in the trace. Or, if the program crashed, the absence of a trace suggests approximately where the crash occurred.

The `StringBackwards3` program shown earlier used a recursive algorithm to reverse the characters in a string. `StringBackwards4` is the same program, with program traces added (see Figure 2).

```
 1  import java.io.*;
 2
 3  public class StringBackwards4
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        BufferedReader stdin =
 8           new BufferedReader(new InputStreamReader(System.in), 1);
 9
10        System.out.print("Enter a message string: ");
11        String s = stdin.readLine();
12        s = backwards(s);
13        System.out.print(s);
14     }
15
16     public static String backwards(String s)
17     {
18        if (s.length() == 1)
19        {
20           System.out.println("Length = 1, Return: " + s);
21           return s;
22        }
23        else
24        {
25           System.out.println("Length = " + s.length()
26                       + ", Call argument: "
27                       + s.substring(1, s.length()));
28           s = backwards(s.substring(1, s.length())) + s.substring(0, 1);
29           System.out.println("s = " + s);
30           return s;
31        }
32     }
33  }
```

Figure 2. `StringBackwards4.java`

A sample dialogue with this program follows:

```
PROMPT>java StringBackwards4
Enter a message string: hyppopotamus
Length = 12, Call argument: yppopotamus
Length = 11, Call argument: ppopotamus
Length = 10, Call argument: popotamus
Length = 9, Call argument: opotamus
Length = 8, Call argument: potamus
Length = 7, Call argument: otamus
Length = 6, Call argument: tamus
Length = 5, Call argument: amus
Length = 4, Call argument: mus
Length = 3, Call argument: us
Length = 2, Call argument: s
Length = 1, Return: s
s = su
s = sum
s = suma
s = sumat
s = sumato
s = sumatop
s = sumatopo
s = sumatopop
s = sumatopopp
s = sumatopoppy
s = sumatopoppyh
sumatopoppyh
```

Program traces are added in lines 20, 25-27, and 29. Not only do program traces help in debugging, they can reveal the inner behaviour of complex algorithms, in this case a recursive algorithm. The output above clearly shows the deepening recursion as method calls are nested inside previous calls, as well as the unwinding of the recursion.

Test Cases

It is a pleasant relief to write a program and get it working properly. You have invested many hours crafting the code — fixing bugs along the way — and when done, it's a great feeling. But, how do you know your code works properly? That's easy, you tested the program and it performed the intended task and generated the correct results. End of story. Hopefully it is, but sometimes an odd thing happens. Many days or weeks later you are working on a new program that uses methods and classes from your earlier work. All of a sudden you notice that results aren't quite as expected. Upon further inspection, you discover that the problem lies in a method or class from the earlier program.

What went wrong in the preceding scenario? It all lies in the testing. We are in the habit of testing our programs with "nominal" data or input conditions, and we are sometimes too quick to pat ourselves on the back when a program works. As it turns out, finding a comprehensive set of test cases for computer programs is quite a challenge. In fact, for very large software projects combining numerous modules written by teams of programmers, it is an absolutely horrendous task. If the software is destined for safety-critical applications (e.g., nuclear power plants or medical equipment), the stakes are high and validation is a massive and complex undertaking.

We needn't worry here, because we just want to write and debug some simple Java programs. However, a little extra effort can go along way in creating robust code. When testing portions of code (i.e., methods), devise test cases that cover the range of possible conditions that may arise. Nominal cases are fine, but test for the extremes as well. For any method that receives an

argument, consider, and test for, extreme cases. If a string argument is passed, make sure the method smoothly handles an empty string. If an integer or double argument is passed, make sure the method can cope with negative values, or special values like zero. These are known as *precondtions*, and they should be tested for at the beginning of a method.

If a method performs numeric calculations, try to anticipate the possibility of results that are out of range. If an integer result is greater than 2^{31}, it cannot be represented by an `int`. A `long` variable should be used. If the result is greater than 2^{63}, it cannot be represented by a `long`. At this point you may have a task for which Java is ill-suited, but, at the very least, attempt to anticipate and accommodate these possibilities.[1]

Let's take this discussion a little further by re-examining a method presented earlier in two programs. The program `Factorial` included a non-recursive method named `factorial()`. The program `Factorial2` included a recursive version of the same method. We noted earlier that the two programs "behaved" the same, and this is true. The following is a sample dialogue for either program, showing an attempt to compute the factorial of a negative number:

```
PROMPT>java Factorial
Enter a non-negative integer: -3
Oops!
```

Both programs reasonably anticipate the possibility of the user inputting a negative number. But, there is a problem lurking if one attempts to paste the code for either `factorial()` method into another program. The check for a negative value occurred "outside" the method. That is, if the value is negative the method is not called in the first place. So, we know how the *program* responds to a negative input, but we don't know how the *method* responds. And, as it turns out, the two methods are quite different in the way they respond to a negative argument. The `factorial()` method in `Factorial` returns the original argument if it is negative (see lines 25-28 in `Factorial`). The `factorial()` method in `Factorial2` embarks on an arduous and incorrect set of recursions if it receives an negative argument (see lines 23-26 in `Factorial2`).[2] This fault was not caught during testing, because the method was not called if the inputted value was negative. A modified version of the recursive `factorial()` method appears below:

```
public static int factorial(int n)
{
   if (n < 0)
      return n;
   if (n == 0)
      return 1;
   else
      return n * factorial(n - 1);
}
```

Now, the method returns immediately with the original argument if it is negative, as with the non-recursive version. This is a precondition and it is checked for at the beginning of the method. The behaviour is consistent and predictable. Yes, but it is still wrong. We cannot calculate the

---

[1] All is not lost, however, as there is a class called `BigInteger` in the `java.math` package that supports integer arithmetic with arbitrary precision.

[2] If the check for negative input is removed in `Factorial2`, and a negative value is entered, the program will crash with a `StackOverflowError`.

factorial of a negative number, just as we cannot calculate the square root of a negative number. By returning the original (negative!) argument we are opening the door to future problems, since we do not know, and cannot anticipate, how a calling program might use the value returned by the `factorial()` method. A more appropriate way to cope with a negative argument is to throw an exception and thereby shift the responsibility (of passing an the invalid argument) back to the calling program. We will learn how to do this later.

While we are discussing the factorial programs, here's something else to consider. What is the factorial of 15? Compare the results from the Factorial program with that obtained using a pocket calculator. We'll leave it to you to think about the different results and to explore these through the debugging techniques just discussed.

Pass by Reference vs. Pass by Value

Most methods are passed arguments when they are called. An argument may be a constant or a variable. For example, in the expression

```
Math.sqrt(33)
```

the constant `33` is passed to the `sqrt()` method of the `Math` class. In the expression

```
Math.sqrt(x)
```

the variable `x` is passed. This is simple enough, however there is an important but simple principle at work here. If a variable is passed, the method receives a copy of the variable's value. The value of the original variable cannot be changed within the method. This seems reasonable because the method only has a copy of the value; it does not have access to the original variable. This process is called *pass by value*.

However, if the variable passed is an object, then the effect is different. This idea is explored in detail in this section. But first, let's clarify the terminology. For brevity, we often say things like, "this method returns an object ...", or "this method is passed an object as an argument ...". As noted earlier, this not quite true. More precisely, we should say,

this method returns a reference to an object ...

or

this method is passed a reference to an object as an argument ...

In fact, objects, per se, are never passed to methods or returned by methods. It is always "a reference to an object" that is passed or returned.

The term "variable" also deserves clarification. There are two types of variables in Java: those that hold the value of a primitive data type and those that hold a reference to an object. A variable that holds a reference to an object is an "object variable" — although, again, the prefix "object" is often dropped for brevity.

Returning to the topic of this section, *pass by value* refers to passing a constant or a variable holding a primitive data type to a method, and *pass by reference* refers to passing an object variable to a method. In both cases a copy of the variable is passed to the method. It is a copy of the "value" for a primitive data type variable; it is a copy of the "reference" for an object variable. So, a method receiving an object variable as an argument receives a copy of the reference to the original object. Here's the clincher: If the method uses that reference to make changes to the object, then the original object is changed. This is reasonable because both the original reference and the copy of the reference "refer to" to same thing — the original object.

There is one exception: strings. Since `String` objects are immutable in Java, a method that is passed a reference to a `String` object cannot change the original object. This distinction is also brought out in this section.

Let's explore pass by reference and pass by value through a demo program (see Figure 1). As a self-test, try to determine the output of this program on your own before proceeding. If understood the preceding discussion and if you guessed the correct output, you can confidently skip the rest of this section. The print statement comments include numbers to show the order of printing.

```
 1  public class DemoPassByReference
 2  {
 3     public static void main(String[] args)
 4     {
 5        // Part I - primitive data types
 6        int i = 25;
 7        System.out.println(i);                          // print it (1)
 8        iMethod(i);
 9        System.out.println(i);                          // print it (3)
10        System.out.println("-----------------");
11
12        // Part II - objects and object references
13        StringBuffer sb = new StringBuffer("Hello, world");
14        System.out.println(sb);                         // print it (4)
15        sbMethod(sb);
16        System.out.println(sb);                         // print it (6)
17        System.out.println("-----------------");
18
19        // Part III - strings
20        String s = "Java is fun!";
21        System.out.println(s);                          // print it (7)
22        sMethod(s);
23        System.out.println(s);                          // print it (9)
24     }
25
26     public static void iMethod(int iTest)
27     {
28        iTest = 9;                                      // change it
29        System.out.println(iTest);                      // print it (2)
30     }
31
32     public static void sbMethod(StringBuffer sbTest)
33     {
34        sbTest = sbTest.insert(7, "Java ");             // change it
35        System.out.println(sbTest);                     // print it (5)
36     }
37
38     public static void sMethod(String sTest)
39     {
40        sTest = sTest.substring(8, 11);                 // change it
41        System.out.println(sTest);                      // print it (8)
42     }
43  }
```

Figure 1. `DemoPassByReference.java`

This program generates the following output:

```
25
9
25
-----------------
Hello, world
Hello, Java world
Hello, Java world
-----------------
Java is fun!
fun
Java is fun!
```

`DemoPassByReference` is organized in three parts. The first deals with primitive data types, which are passed by value. The program begins by declaring and `int` variable named `i` and initializing it with the value 25 (line 6). The memory assignment just after line 6 is illustrated in Figure 2a. The value of `i` is printed in line 7.

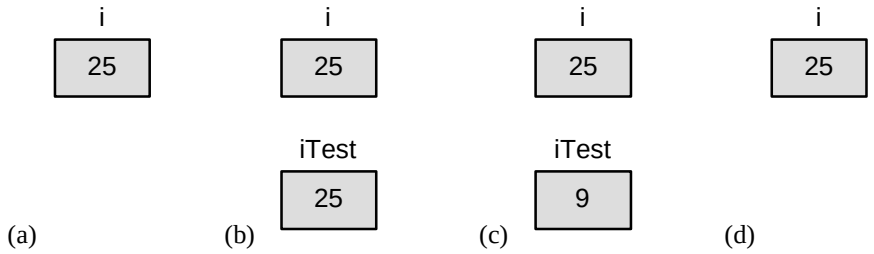


Figure 2. Memory assignments for call-by-value example (`int` variables)

Then, the `int` variable is passed as an argument to a method called `iMethod()` (line 8). The method is defined in lines 26-30. Within the method, a copy of `i` exists as a local variable named `iTest`. The memory assignments just after `iMethod()` begins execution are shown in Figure 2b. Note that `i` and `iTest` are distinct: They are two separate variables that happen to hold the same value. Within the method, the value is changed to 9 (line 28) and then printed again. Back in the `main()` method the original variable is also printed again (line 9). Changing the passed value in `iMethod()` had no effect on the original variable, and, so, the third value printed is the same as the first.

Part II of the program deals with objects and object references. The pass-by-reference concept is illustrated by the object variables `sb` and `sbTest`. In the `main()` method, a `StringBuffer` object is instantiated and initialized with "`Hello, world`" and a reference to it is assigned to the `StringBuffer` object variable `sb` (line 13).

The memory assignments for the object and the object variable are illustrated in Figure 3a. This corresponds to the state of the `sb` just after line 13 in Figure 1. The object is printed in line 14. In line 15, the `sbMethod()` method is called with `sb` as an argument. The method is defined in lines 32-36. Within the method, a copy of `sb` exists as a local variable named `sbTest`. The memory assignments just after the `sbMethod()` begins execution are shown in Figure 3b. Note that `sb` and `sbTest` refer to the same object.

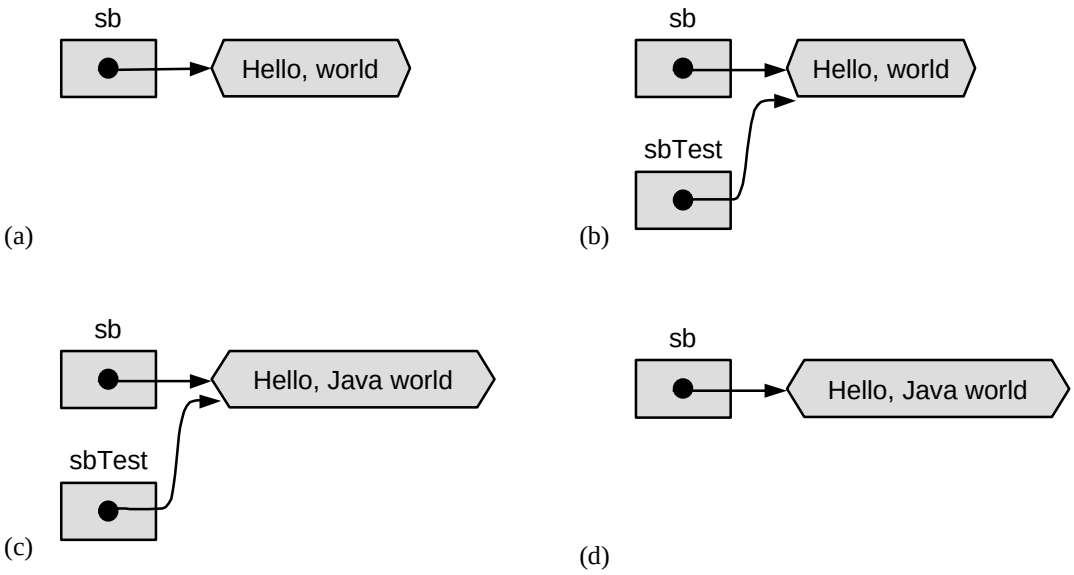


Figure 3. Memory assignments for call-by-reference example (`StringBuffer` objects)

In line 34, the object is modified using the `insert()` method of the `StringBuffer` class. The method is called through the instance variable `sbTest`. The memory assignments just after line 34 executes are shown in Figure 3c. After the `sbMethod()` finishes execution, control passes back to the `main()` method and `sbTest` ceases to exist. The memory assignments just after line 15 in Figure 1 are shown in Figure 3d. Note that the original object has changed, and that the original object variable, `sb`, now refers to the updated object.

The scenario played out above, is a typical example of pass by reference. It demonstrates that methods can change the objects instantiated in other methods when they are passed a reference to the object as an argument. A similar scenario could be crafted for objects of any of Java's numerous classes. The `StringBuffer` class is a good choice for the example, because `StringBuffer` objects are tangible, printable entities. Other objects are less tangible (e.g., objects of the `Random` class); however, the same rules apply.

Part III of the program deals with strings. The `String` class is unlike other classes in Java because `String` objects, once instantiated, cannot change. For completeness, let's walk through the memory assignments for the `String` objects in the `DemoPassByRefererнce` program. A `String` object is instantiated in line 20 and a reference to it is assigned to the `String` object variable `s`. The memory assignments at this point are shown in Figure 4a.

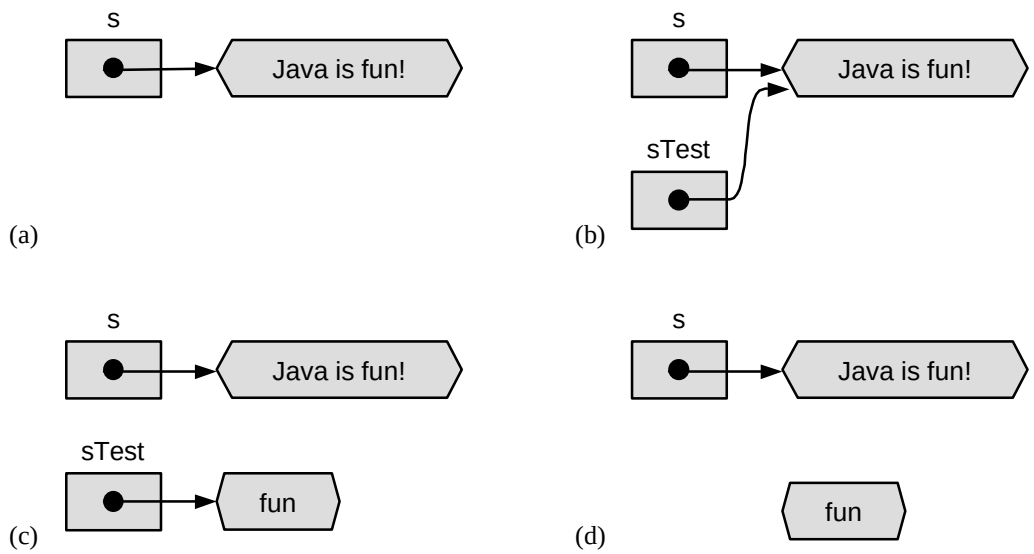


Figure 4. Memory assignments for call-by-reference example (`String` objects)

In line 22 the `sMethod()` is called with `s` as an argument. The method is defined in lines 38-42. Within the method, a copy of `s` exists as a local variable named `sTest`. The memory assignments just after the `sMethod()` begins execution are shown in Figure 4b. Note that `s` and `sTest` refer to the same object. At this point, there is no difference in how memory was assigned for the `String` or `StringBuffer` objects. However, in line 40, Java's unique treatment of `String` objects surfaces. The `substring()` method of the `String` class is called to extract the string "fun" from the original string. The result is the instantiation of a new `String` object. A reference to the new object is assigned to `sTest`. The memory assignments just after line 37 executes are shown in Figure 4c.

After the `sMethod()` finishes execution, control passes back to the `main()` method and `sTest` ceases to exist. The memory assignments just after line 22 in Figure 1 are shown in Figure 4d. Note that the original object did *not* change. The `String` object containing "fun" is now redundant and its space is eventually reclaimed.

Variable Scope - Revisited

We discussed variable scope earlier in our study of the `while`, `do/while`, and `for` loops. Now that we are defining and using custom methods, a few more points are warranted. The general rule is that a variable has scope, or visibility, from the point where it is declared to the end of the block where it is declared, and examples were given in Chapter 3.

When defining a method, however, it sometimes happens that a programmer declares a variable and picks a name for it that matches a variable name in another method. It is also possible that one of these methods calls the other. When this happens, the variables are treated a separate variables. Figure 1 shows a code segment including a `main()` method and a method called `testMethod()`. Both methods include the declaration of `int` variable named `i`. In each case, the scope of the variable is shown in a thick line.

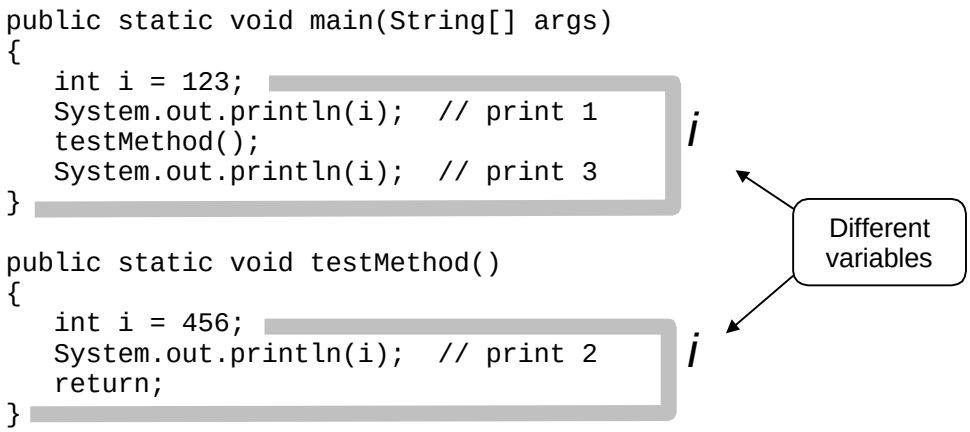


Figure 1. Variable scope revisited

Within the `main()` method, `testMethod()` is called. A variable `i` is declared and assigned 456 in `testMethod();` however, this has no influence on the variable `i` in `main()`. Three print statements appear and their sequence of execution is numbered. The output generated is

```
123
456
123
```

The third line above emphasizes that the assignment of the variable `i` in `testMethod()` did not change the content of the variable with the same name in `main()`.

Key Points

To this point, we have learned how to write reasonably large Java programs by defining and using custom methods. Here are some key points we have learned:

- Defining custom methods allows us to (i) cope with complexity, (ii) hide low-level details, and (iii) reuse portions of code.
- A "public" method is accessible anywhere the class in which it is defined is accessible.
- There are three type of methods: constructors, class methods, and instance methods.
- A "class" method is also called a "static" method.
- A class method is not called through an instance variable.
- An example of a class method is the `sqrt()` method in the `Math` class.
- An "instance" method is called through an instance of a class — an object variable.
- An example of an instance method is the `substring()` method of the `String` class.
- Custom-defined methods can be used to improve the formatting of output data.
- Recursion is a technique whereby a method calls itself.
- Program traces are an effective debugging tool.
- When a primitive data type variable is passed to a method, it is *passed by value*.
- When an object is passed to a method, it is *passed by reference*.
- A method that receives a primitive data type variable as an argument *cannot* change the value of the original variable.
- A method that receives an object variable as an argument *can* change the content of the original object.

Chapter 6

Arrays and Vectors

Arrays

We will now explore different ways to organize and access data. Let's begin with arrays. An *array* is a collection of variables of the same type. An array of integers named `x` is declared as follows:

```
int[] x;
```

The set of square brackets identifies `x` as an integer array, as opposed to a simple integer variable. However, this statement does not set aside space for the array. Space is allocated as follows:

```
x = new int[6];
```

The two statements above are often combined:

```
int[] x = new int[6];
```

The reserved word "`new`" is familiar territory, as it precedes calls to constructor methods. The number "6" above is the number of elements in the array. The effect of the statement is to declare `x` as an integer array containing six elements. Each element is automatically initialized with the value 0. To place a different value in an element of the array, a statement such as the following may be used:

```
x[0] = 500;
```

This statement places the integer value 500 in the 0^{th} element of the array. The number "0", above, is an array *index*, also called an array *subscript*.

With this introduction, let's proceed to an example. The program `DemoArray` uses an array to store and output the squares of integers from 0 to 5 (see Figure 1).

```
 1  public class DemoArray
 2  {
 3     public static void main(String[] args)
 4     {
 5        int[] numbers = new int[6];
 6        numbers[0] = 0;
 7        numbers[1] = 1;
 8        numbers[2] = 4;
 9        numbers[3] = 9;
10        numbers[4] = 16;
11        numbers[5] = 25;
12        for (int i = 0; i < 6; i++)
13           System.out.println(i + "  " + numbers[i]);
14     }
15  }
```

Figure 1. `DemoArray.java`

This program generates the following output:

```
0  0
1  1
2  4
3  9
4  16
5  25
```

A six-element integer array named `numbers` is declared in line 5. Lines 6 through 11 initialize `numbers` with the squares of integers 0 through 5. Lines 12-13 output the contents of the array to the standard output. The loop control variable, `i`, is used within the `for` loop as an index into the array to retrieve elements in the array.

The memory assignments for `numbers` at two critical places in the program are shown in Figure 2. Just after line 5, each element in the array contains 0 as a default value, as shown in Figure 2a. Over the next six lines, each element is re-assigned a new value. The final state of the array, just after line 11, is shown in Figure 2b.

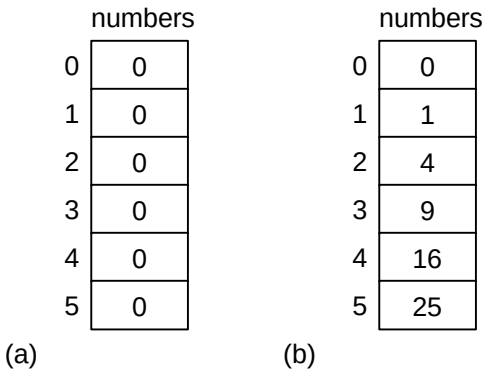


Figure 2. Memory assignments in `DemoArray` (a) after line 5 (b) after line 11

The figure shows the array elements in boxes and the array indices to the left of the boxes.

One of the most common errors in working with arrays is attempting to access a non-existing element. For example, let's assume the following line is added just after line 11 in `DemoArray`:

```
numbers[6] = 36;
```

The modified program will compile without errors. However, when it is executed, a run-time error occurs and the following message appears at the standard error stream (the host system's CRT display):

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
   at DemoArray.main(Compiled Code)
```

Ouch! The critical term above is "`ArrayIndexOutOfBoundsException`". An attempt was made to assign a value to an array element that did not exist. The array `numbers` contains six elements, but the indices range from 0 to 5. The index 6 is *out of bounds*.

The length Field

Knowing the length of an array is so critical that a public data field is available for any array. The name of the field is `length`. The length of the `numbers` array in `DemoArray`, for example, is

```
numbers.length
```

It equals 6 because the `numbers` array has 6 elements. An excellent use of `length` is in setting up a `for` loop. Line 12 in `DemoArray` could be replaced with

```
for(int i = 0; i < numbers.length; i++)
```

Instead of explicitly entering the constant "6", the length is specified using the `length` field of the array. This is much safer, since changes to the size of the array in the source program are automatically accommodated when the program is re-compiled and executed. Note that the `for` loop does not execute for `i` = 6. The expression `i < numbers.length` ensures that the loop only executes up to `i` = 5.

Here's an important caution. Don't confuse `length` — the length of an array — with the `length()` method of the `String` class. They perform similar operations, but they work with different entities in the language. For example, if line 12 in `DemoArray` is replaced with

```
for (int i = 0; i < numbers.length(); i++)  // Wrong!
```

you will be greeted by the following message when the program is compiled:

```
DemoArray.java:24: Method length() not found in class java.lang.Object.
      for (int i = 0; i < numbers.length(); i++)   // Wrong!
                                          ^
1 error
```

This is very definitely a syntax error, as the `length()` method of the `String` class cannot be used with an array object. The error is caught by the compiler.

Initialization Lists

It is possible to initialize array elements without the step-by-step series of assignments in `DemoArray`. Instead, an *initialization list* can be used. For example, lines 5-11 in `DemoArray` could be replaced with the following single statement:

```
int[] numbers = { 0, 1, 4, 9, 16, 25 };
```

The effect is to declare `numbers` as an integer array and initialize it with the values enclosed in braces. Note that a comma follows each value except the last and that the closing brace is followed by a semicolon.

In previous demo programs, we used both variables and constants. Variables change as a program executes, constants do not. When an initialization list is used, it is often the case that the array holds constants. Strictly speaking, such arrays should be declared "`final`" as with other variable constants, thus ensuring elements are not inadvertently changed during program execution. Let's illustrate this with a simple demo program using an initialization list. The program `CubeIt` prompts the user for an integer from 1 to 10 and then outputs the cube of the integer (see Figure 3).

```
 1  import java.io.*;
 2
 3  public class CubeIt
 4  {
 5     private static final int[] CUBE =
 6        { 0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000 };
 7
 8     public static void main(String[] args) throws IOException
 9     {
10        BufferedReader stdin =
11           new BufferedReader(new InputStreamReader(System.in), 1);
12
13        System.out.print("Enter an integer from 0 to " +
14           (CUBE.length - 1) + ": ");
15
16        int n = Integer.parseInt(stdin.readLine());
17
18        if (n >= 0 && n < CUBE.length)
19           System.out.println("It's cube is " + CUBE[n]);
20        else
21           System.out.println("Value out of range!");
22     }
23  }
```

Figure 3. `CubeIt.java`

A sample dialogue with this program follows: (User input is underlined.)

```
PROMPT>java CubeIt
Enter an integer from 0 to 10: 9
It's cube is 729
```

An array named `CUBE` is declared and initialized (using an initialization list) in lines 5-6. Note that the name of the array is set in uppercase characters, as consistent with the naming conventions for Java constants (see Chapter 2). The array is positioned before the `main()` method, but it could just as easily go after it.

The value entered by the user is converted from a `String` to an integer in line 16 and assigned to the `int` variable `n`. This variable is then used in line 19 as an index into `CUBE` to retrieve the cube of the value entered.

The `length` field is used in an interesting way in `CubeIt`. The initialization list in line 6 contains 11 integers, representing the cubes of integers from 0 to 10 inclusive. However, the program does not make a numeric reference to the length of the array. Instead, `CUBE.length` is used (lines 14 and 18). This program could be modified to include more (or fewer) entries in the array simply by editing the initialization list. The check for correct user input in line 19 is adjusted automatically when the program is re-compiled. Furthermore, the prompt string outputted in lines 13-14 is also adjusted automatically. Since string concatenation is used, the integer expression `(CUBE.length - 1)` appears as its string-equivalent in the prompt ("`10`" in this case).

Arrays of Other Data Types

Thus far, we have discussed only integer arrays. Of course, arrays are a legitimate data structure for any of Java's eight primitive data types, as well as for objects. (We will discuss arrays of objects shortly.) All issues discussed above for the `int` data type apply in the same manner for

the `char`, `byte`, `short`, `long`, `float`, `double`, and `boolean` data types. The only difference is in the default values in the event an array is declared but not initialized. For integer or floating point arrays (`byte`, `short`, `int`, `long`, `float`, and `double`), the array elements are initialized with zero. For `boolean` arrays and for `char` arrays, the array elements are initialized with `false` and with null characters, respectively.

An example of a character array follows:

```
char[] greeting = { 'h', 'e', 'l', 'l', 'o' };
```

A `boolean` array might be useful, for example, to hold the status of spaces on a game board. The *Tic Tac Toe* game has nine spaces that are empty when a game begins. As a game progress, the spaces are gradually filled with X's or O's. An array holding the status of the spaces could be declared as follows:

```
boolean[] spaceOccupied = new boolean[9];
```

All nine elements of the array `spaceOccupied` are initialized by default with `false`. As the game progress, elements can be assigned `true` as spaces are selected.

Arrays of Objects

Java supports arrays of objects in a manner consistent with that for primitive data types. If a class existed for `Widget` objects, then an array named `fasteners` holding 100 such objects could be declared as follows:

```
Widget[] fasteners = new Widget[100];
```

In the preceding section, we identified the default values for array elements when an array is declared but is not explicitly initialized. When the declaration is for an array of objects, the default value is a null reference. So the statement above allocates memory for 100 references to `Widget` objects and each reference is initialized with `null`.

Let's recast the above in terms of the objects most familiar to us: strings. The following declares an array of `String` objects named `sea` of size 4:

```
String[] sea = new String[4];
```

This array initially contains four null references. We can initialize the array as follows:

```
sea[0] = "Mediterranean";
sea[1] = "Black";
sea[2] = "North";
sea[3] = "Red";
```

Or, the five preceding statements can be combined using an initialization list:

```
String[] sea = { "Mediterranean", "Black", "North", "Red" };
```

The memory assignments after the above statement executes are shown in Figure 4. Strictly speaking, the array `sea` holds an array of references, not objects. (The objects are somewhere else.) Nevertheless, it is customary to refer to such an array as "an array of objects", with the understanding that the memory assignments are as shown in Figure 4.

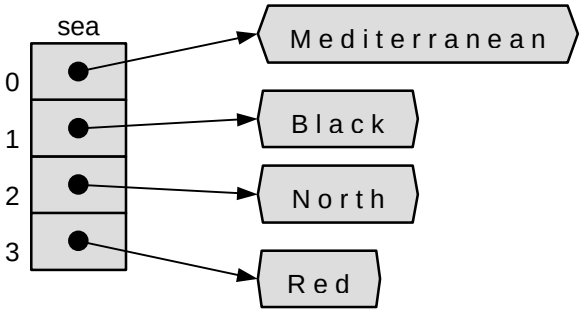


Figure 4. Memory assignments for an array of `String` objects

The elements of the array could be printed as follows:

```
for (int i = 0; i < sea.length; i++)
   System.out.println(sea[i]);
```

Here is a reminder of the difference between `length` and `length()`. Note that `sea.length` is the length of the array (in this case, 4), whereas `sea[3].length()` is the length of the 4th string in the array (in this case, 3). Either of the alternate expressions `sea.length()` or `sea[3].length` is invalid and will cause a compile error.

Let's put these ideas to work in a demo program. The program `DemoStringArray` counts and outputs some of Java's reserved words (see Figure 5).

```
 1  public class DemoStringArray
 2  {
 3     private static final String[] RESERVED_WORDS = {
 4           "abstract", "boolean", "break", "byte", "byvalue", "case",
 5           "cast", "catch", "char", "class", "const", "continue",
 6           "default", "do", "double", "else", "extends", "false",
 7           "final", "finally", "float", "for", "future", "generic",
 8           "goto", "if", "implements", "import", "inner", "instanceof",
 9           "int", "interface", "long", "native", "new", "null",
10           "operator", "outer", "package", "private", "protected",
11           "public", "rest", "return", "short", "static", "super",
12           "switch", "synchronized", "this", "throw", "throws",
13           "transient", "true", "try", "var", "void", "volatile", "while"
14        };
15
16     public static void main(String[] args)
17     {
18        int n = RESERVED_WORDS.length;
19        System.out.println("Java has " + n + " reserved words");
20        System.out.println("The reserved words beginning with 's' are");
21        for (int i = 0; i < n; i++)
22           if (RESERVED_WORDS[i].charAt(0) == 's')
23              System.out.println(RESERVED_WORDS[i]);
24     }
25  )
```

Figure 5. `DemoStringArray.java`

This program generates the following output:

```
Java has 59 reserved words
The reserved words beginning with 's' are
short
static
super
switch
synchronized
```

Since Java's list of reserved words is fixed, the array `RESERVED_WORDS` is declared as "`final`" outside the `main()` method. Within the `main()` method, the first statement retrieves the length of the array and assigns it to the `int` variable `n` (see line 18). In line 19, the number of reserved words is printed. (Java has 59 reserved words.) Then we proceed to print the reserved words (lines 21-23); however, we only print words beginning with 's'. The `if` statement in line 22 uses the following expression:

```
RESERVED_WORDS[i].charAt(0) == 's'
```

This is a relational expression that yields `true` or `false`. If it is `true`, the word at index `i` in the array is printed, otherwise we proceed to check the next word. Since `RESERVED_WORDS` is a array of `String` objects, the term `RESERVED_WORDS[i]` can be used anywhere a `String` variable can be used — such as preceding a `String` method using dot notation. The method `charAt()` is an instance method of the `String` class. With 0 as an argument, it returns the character at index 0 in the string. In this case, it returns the first character in the current word. Connecting this to the constant 's' with a test for equality (`==`) achieves the desired effect of determining if the current word in the array begins with the letter 's'. If so, it is printed.

Arrays of Button and AudioClip Objects

Lest we get too comfortable working mostly with `String` objects, let's explore arrays of other objects. The demo program `DemoSound` is an applet that plays a sound when an on-screen button was selected with a mouse click. Let's extend this idea with an applet that displays ten buttons and plays a different sound for each button. The applet `DemoSoundArray` includes an array of ten `Button` objects and ten `AudioClip` objects (see Figure 6).

```
 1  import java.awt.*;
 2  import java.awt.event.*;
 3  import java.applet.*;
 4
 5  public class DemoSoundArray extends Applet implements ActionListener
 6  {
 7     private static final int MAX = 10;
 8     private AudioClip[] tone = new AudioClip[MAX];
 9     private Button[] beep = new Button[MAX];
10
11     public void init()
12     {
13        for (int i = 0; i < MAX; i++)
14        {
15           String soundFile = "sounds\\" + i + ".au";
16           tone[i] = getAudioClip(getDocumentBase(), soundFile);
17           beep[i] = new Button("Beep " + i);
18           beep[i].addActionListener(this);
19           add(beep[i]);
20        }
21     }
22
23     public void actionPerformed(ActionEvent ae)
24     {
25        for (int i = 0; i < MAX; i++)
26           if (ae.getSource() == beep[i])
27              tone[i].play();
28     }
29  }
```

Figure 6. `DemoSoundArray.java`

When this applet executes (using `appletviewer` or a web browser), the arrangement of buttons shown in Figure 7 appears in the applet window.

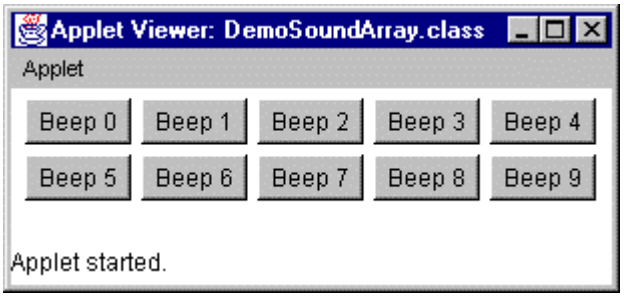


Figure 7. Graphic window from `DemoSoundArray` applet

When a button is selected with a mouse click, a beep is heard.

Once again, we must defer a detailed discussion of some aspects of this program to later chapters. (The implementation of action listeners is presented later.) We are mainly concerned with this applet's use of arrays of objects. Two arrays are declared. In line 9 an array of `AudioClip`

objects named `tone` is declared, and in line 10 an array of `Button` objects named `beep` is declared.[1] The arrays are initialized in lines 16-17.

The `getAudioClip()` method of the `Applet` class gets an audio clip and returns a reference to it. The reference is stored in the `tone` array. Two arguments are required and together they identify the location of the audio clip. The first argument is a URL (Uniform Resource Locator) specifying the location of the document in which the applet is embedded. This is provided by the `getDocumentBase()` method of the `Applet` class. It provides a URL, or a path, to the applet on the host system. The second argument is a string representing the name and location of the audio clip, relative to the URL. In the example, the string is initialized in the preceding line with the expression

```
"sounds\\" + i + ".au"
```

The first time through the loop, `i` equals 0, so the expression equals `sounds\0.au`. The ten audio clips are stored in files named `0.au`, `1.au`, etc. up to `9.au` in a subdirectory named `sounds`. (Note that '`\\`' is the escape sequence representing the backslash character.)

The `Button` array is initialized using the `Button()` constructor. The argument is the button's label represented as a string. As with the audio clip filenames, the button labels are created on the fly using string concatenation. The effect is readily seen in the button labels in Figure 7.

The link between mouse clicks and buttons is established in line 18 by adding an action listener for each button. The buttons are added to the applet window in line 19. At this point, everything is setup and ready to go. Note that the `actionPerformed()` method is not explicitly called in the `init()` method. It is called automatically when a mouse click occurs in a button while the applet is running. Lines 25-27 scan through the `beep` array to determine which button the click is associated with. When a match is found, the tone for that button is played (line 27).

Command Line Arguments

Although this chapter marks our formal introduction to arrays, every Java application seen thus far included an array specification. It appeared in the `main()` method signature:

```
public static void main(String[] args)
```

The specification of arguments passed to `main()` appears within parentheses. Although we have already seen numerous examples of method arguments, this is our first example of an array argument. The signature above identifies `args` as an array of `String` objects. The `String` objects contained in `args` represent the *command-line arguments* entered when the Java program was launched. This is a very simple but powerful feature. Let's see how it works. The program `DemoCommandLineArgs` prints out a count of its command-line arguments as well as the individual arguments (see Figure 8).

---

[1] In fact, `AudioClip` is an *interface* (not a class), but for this discussion it can be thought of as a class. An interface is a collection of constants and abstract methods. The methods of the `AudioClip` interface (e.g., `play()`) are implemented in the `Applet` class.

```
1  public class DemoCommandLineArgs
2  {
3     public static void main(String[] args)
4     {
5        System.out.println(args.length + " command-line argument(s)");
6        for (int i = 0; i < args.length; i++)
7           System.out.println(args[i]);
8     }
9  }
```

Figure 8. `DemoCommandLineArgs.java`

A sample dialogue with this program follows:

```
PROMPT>java DemoCommandLineArgs Have a nice day!
4 command-line argument(s)
Have
a
nice
day!
```

Four arguments were entered, as shown above. Note that the command itself is not considered a command-line argument. The number of command-line arguments is retrieved within the program using `args.length`. This value is printed in line 5. It is also used in line 6 to setup the `for` loop that prints the arguments on separate lines. The arguments are retrieved and printed in line 7 as expected: `args[0]` is the first command-line argument, `args[1]` is the second command-line argument, and so on.

Space and tab characters (`'\t'`) serve as delimiters for the arguments. The arguments can contain any combination of letters, digits, or punctuation characters, with the exception of a double quote. Multiple words are treated as one argument if they are enclosed in double quotes:

```
PROMPT>java DemoCommandLineArgs Canada "United States"
2 command-line argument(s)
Canada
United States
```

The examples above show how command-line arguments work. However, they don't shed light on how command-line arguments can contribute to a Java program. In fact, command-line arguments are an extremely useful mechanism to get extra information into a program without coding it in the source file. The extra information usually modifies the default operation of the program in some manner. For example, at the DOS prompt the following command

```
dir /os
```

displays a directory listing sorted by increasing file size. The string "`/os`" is a command-line argument. Within the `dir` program, this argument, if present, is used to control the format of the directory listing. (For a listing of all options for the `dir` command, enter `dir /?`.)

Some programs may "require" extra information. For example, the DOS `type` command outputs the content of a file to the standard output. If the command is invoked without specifying a file (i.e., the number of arguments is zero), the message "`Required parameter missing`" is printed.

Let's demonstrate the use of command-line arguments to control a program's behaviour. The program `FindString` reads text from the standard input and echoes tokens to the standard output — but only if they contain a string specified in the command line (see Figure 9).

```
 1  import java.io.*;
 2  import java.util.*;
 3
 4  public class FindString
 5  {
 6     public static void main(String[] args) throws IOException
 7     {
 8        // precisely one command-line argument required
 9        if (args.length != 1)
10        {
11           System.out.println("usage: java FindString \"string\"");
12           return;
13        }
14
15        // prepare keyboard for input
16        BufferedReader stdin =
17              new BufferedReader(new InputStreamReader(System.in), 1);
18
19        // process lines until 'null' (no more input)
20        String line;
21        while ((line = stdin.readLine()) != null)
22        {
23           // prepare to tokenize line
24           StringTokenizer st = new StringTokenizer(line);
25
26           // process tokens in line
27           while (st.hasMoreTokens())
28           {
29              String s = st.nextToken();
30              if (s.indexOf(args[0]) >= 0)
31                 System.out.println(s);
32           }
33        }
34     }
35  }
```

Figure 9. `FindString.java`

A sample dialogue with this program follows: (Input is read from a test file named `testdata.txt`.)

```
PROMPT>java FindString "te" < testdata.txt
Microsystems,
redistribute
intended
maintenance
redistribute
```

Only five words in the test data file contain the string "te". The program follows the same general framework as `EchoAlphaWords` in Chapter 4. Lines 9-13 ensure that precisely one command-line argument was entered. If `args.length` does not equal 1, the program is terminated early with the following message:

```
usage: java FindString "string"
```

Note that input is read from the standard input; so, a reference to the input redirection syntax is not warranted in the usage message. The double quotes are removed from the string stored in the array passed on to the `main()` method.

Line 30 contains the critical step. The `indexOf()` method in the `String` class returns an integer representing the position of a substring in a string. The substring is the command-line argument `args[0]`, and the string is a token in the input stream. If the substring is not in the string, -1 is returned. Any return value equal or greater than zero means the substring was found. If so, the string is printed.

As another example of command-line arguments, the program `RandomGen` outputs a series of random floating-point numbers based on three arguments provided on the command line. The arguments specify the number of random numbers to generate and the minimum and maximum of the range for the numbers (see Figure 10).

```
 1  public class RandomGen
 2  {
 3     public static void main(String[] args)
 4     {
 5        // exactly three arguments required
 6        if (args.length != 3)
 7        {
 8           System.out.println("Usage: java RandomGen n min max");
 9           return;
10        }
11
12        // convert command-line arguments
13        int n = Integer.parseInt(args[0]);
14        double min = Double.parseDouble(args[1]);
15        double max = Double.parseDouble(args[2]);
16
17        // generate the random numbers
18        for (int i = 0; i < n; i++)
19        {
20           double r = Math.random() * (max - min) + min;
21           System.out.println(r);
22        }
23     }
24  }
```

Figure 10. `RandomGen.java`

As sample dialogue with this program follows:

```
PROMPT>java RandomGen 10 5.5 7.0
6.09335317632495
6.653924534818176
5.592448680647347
5.886007854214609
6.832437072923562
5.6293184223093045
6.398294060647076
6.795585314105339
6.825760402283336
5.566488584867678
```

We'll use `RandomGen` later to generate test data for other programs. For the moment, let's just examine how command-line arguments are used to control the program's behaviour. In the sample dialogue, three arguments appear: 10 (the number of random floating-point numbers to generate) and 5.5 and 7.0 (the range for the numbers). The output clearly shows ten numbers lying in this range.

The program requires exactly three command-line arguments. If `args.length` is not three, the program terminates early with a usage message. The three arguments are converted, as appropriate, in lines 13-15 and assigned to the variables `n`, `min`, and `max` respectively. The numbers are generated in a `for` loop that executes `n` times. Within the loop a random double is generated in line 20 using the expression

```
Math.random() * (max - min) + min
```

Since the `random()` method generates a random `double` in the range 0.0 to 1.0, multiplying by `max - min` and then adding `min` effectively scales the result into the desired range. The number is printed in line 21.

Finally, let's combine command-line arguments with our discussion of file streams in Chapter 4. The program `JavaType` mimics the `DOS TYPE` command (see Figure 11).

```
 1  import java.io.*;
 2
 3  public class JavaType
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        if (args.length < 1)
 8        {
 9           System.out.println("Required parameter missing");
10           return;
11        }
12        else if (args.length > 1)
13        {
14           System.out.println("Too many parameters");
15           return;
16        }
17
18        File f = new File(args[0]);
19        if (!f.exists())
20        {
21           System.out.println("File not found - " + args[0]);
22           return;
23        }
24
25        // open disk file for input
26        BufferedReader inputFile =
27           new BufferedReader(new FileReader(args[0]));
28
29        // read lines from the disk file, write lines to console
30        String s;
31        while ((s = inputFile.readLine()) != null)
32           System.out.println(s);
33
34        // close disk file
35        inputFile.close();
36     }
37  }
```

Figure 11. `JavaType.java`

This program requires one command-line argument specifying the name of a file to "type" to the standard output. If the number of command-line arguments is less than one, the message "Required parameter missing" is output (lines 7-11). If the number of command-line arguments is greater than one, the message "Too many parameters" is output (lines 12-16). If precisely one argument was entered, yet no corresponding file exists, the message "File not found - *filename*" is output, where *filename* is the command-line argument (lines 18-23). If we get past these three checks, we're in the clear. The file is opened as a `BufferedReader` object named `inputFile` in lines 26-27, its contents are read and sent to the standard output in lines 30-32, then the file is closed in line 35.

Working With Arrays

The term "number crunching" is computer-talk for the sorts of operations we do on large volumes of data. The data are usually stored in disk files and are read into a program for "crunching" — for analysis in various ways. Let's see how arrays are used for typical number crunching operations. The program `NumberStats` reads data into an array, then calculates some summary statistics on the data (see Figure 12).

```
 1  import java.io.*;
 2  import java.util.*;
 3
 4  public class NumberStats
 5  {
 6     private static final int MAX = 100; // maximum number of numbers
 7
 8     public static void main(String[] args) throws IOException
 9     {
10        BufferedReader stdin =
11           new BufferedReader(new InputStreamReader(System.in), 1);
12
13        double[] numbers = new double[MAX];
14        int size = 0;
15
16        // input data and put into array
17        String line;
18        while ((line = stdin.readLine()) != null)
19        {
20           // prepare to tokenize line
21           StringTokenizer st = new StringTokenizer(line, " ,\t");
22
23           // process tokens in line
24           while (st.hasMoreTokens())
25           {
26              String s = st.nextToken();
27              numbers[size] = Double.parseDouble(s);
28              size++;
29           }
30        }
31
32        // output some statistics on the data
33        System.out.println("N = " + size);
34        System.out.println("Minimum = " + min(numbers, size));
35        System.out.println("Maximum = " + max(numbers, size));
36        System.out.println("Mean = " + mean(numbers, size));
37        System.out.println("Standard deviation = " + sd(numbers, size));
38     }
39
40     // find the minimum value in an array
41     public static double min(double[] n, int length)
42     {
43        double min = n[0];
44        for (int j = 1; j < length; j++)
45           if (n[j] < min)
46              min = n[j];
47        return min;
48     }
49
50     // find the maximum value in a array
51     public static double max(double[] n, int length)
52     {
53        double max = n[0];
54        for (int j = 1; j < length; j++)
55           if (n[j] > max)
56              max = n[j];
57        return max;
58     }
59
60     // calculate the mean of the values in an array
61     public static double mean(double n[], int length)
62     {
```

```
63         double mean = 0.0;
64         for (int j = 0; j < length; j++)
65            mean += n[j];
66         return mean / length;
67      }
68
69      // calculate the standard deviation of values in an array
70      public static double sd(double[] n, int length)
71      {
72         double m = mean(n, length);
73         double t = 0.0;
74         for (int j = 0; j < length; j++)
75            t += (m - n[j]) * (m - n[j]);
76         return Math.sqrt(t / (length - 1.0));
77      }
78
79   }
```

Figure 12. `NumberStats.java`

Although the program works fine with keyboard entry, let's show it in action with data read from a file. For this, a sample file was created with stock market trading data for the *euro* — the currency in Europe introduced at the beginning of 1999. Each entry is the euro's value in US dollars, with a sample recorded every hour for 24 hours, beginning 6:00 p.m. Eastern Standard Time on January 3, 1999, just after the euro's introduction. The content of the file is shown in Figure 13.

```
1.175, 1.176, 1.179, 1.182, 1.188, 1.185, 1.187, 1.188,
1.186, 1.184, 1.180, 1.180, 1.179, 1.180, 1.178, 1.180,
1.180, 1.180, 1.182, 1.181, 1.182, 1.182, 1.183, 1.183
```

Figure 13. Content of `euro.dat` sample data file (source: *Globe and Mail*)

Although we could easily design `NumberStats` to read data from a file specified on the command, the program is setup to read from the standard input. The following is a sample dialog of the program inputting data from `euro.dat` using input redirection:

```
PROMPT>java NumberStats < euro.dat
N = 24
Minimum = 1.175
Maximum = 1.188
Mean = 1.1816666666666664
Standard deviation = 0.0034220089023169176
```

During the 24 hour period observed, the euro's value in US dollars ranged from $1.175 to $1.188, with a mean of $1.182 and a standard deviation of $0.003422 ($n$ = 24).

The program includes four methods in addition to `main()`:

`min()`	find the minimum value in an array (lines 41-48)
`max()`	find the maximum value in an array (lines 51-58
`mean()`	calculate the mean of the values in an array (lines 61-67)
`sd()`	calculate the standard deviation of the values in an array (lines 70-77)

The `min()` and `max()` methods are straight forward. The `mean()` and `sd()` methods are based on formulas found in any statistics book:

$$mean = \overline{X} = \frac{\sum x_i}{n} \qquad sd = \sqrt{\frac{\sum (\overline{X} - x_i)^2}{n-1}}$$

So, the mean is calculated by summing all the array elements, then dividing by the number of elements. The standard deviation is calculated by subtracting each element from the mean, squaring the result, repeating this for each element in the array, summing the results over all elements in the array, dividing the result by "one less than" the number elements, and then taking the positive square root of the result. Now, if you re-read the preceding sentence, it might actually make sense to you. But, perhaps you should just compare the formula above and with lines 70-77 in `NumberStats`. With our understanding of loops, methods, and arrays, it is apparent that the `sd()` method caclulates the standard deviation of elements in an array.

In the `main()` method we see how the statistics methods are put to use. The data for this program originate external to the program, and the number of items of data to be processed is not known. This creates a problem. Once an array is declared in Java, its size it fixed. Individual elements can change, but the size cannot. For this reason, a Java array is called a *static data structure*. (The term "static", in this sense, has no connection the formal use of the same term in the Java lanaguage.) *Dynamic data structures* are supported through the `Vector` class, and we'll see examples of these shortly.

To get around the "fixed-size" problem, an approach is taken in `NumbersStats` that is not particularly elegent. But it works. A final constant named `MAX` is defined in line 6 and set to 100. This program can process up to 100 data items. To process more data, we must change the source code and recompile. A `double` array named `numbers` is declared in line 13. Its size is 100 elements. So, the expressionn `numbers.length` is an integer equal to 100. This isn't much use in this program, because we aren't reading 100 numbers. We have no option but to declare and maintain a local variable that represents the "actual" number of elements placed in the array. The variable `size` serves this purpose. It is declared and initialized to zero in line 14. As data are read, `size` is incremented (line 28). Anywhere the "length" of the array is required for subsequent operations, it is impertive that `size` is used, not `numbers.length`. In fact, all the statistical methods were defined to received an integer length as an argument. This is cumbersome, but we really have no option. If the methods used the "`.length`" field of the array argument, the results would be wrong because portions of the array never initialized would be included in the statistics.

The data in `euro.dat` are formatted in three lines with eight values per line. The values are followed by a comma and a space character, and by a newline at the end of each line. This format is convenient, but it is not "required". If the data were organized one item per line, or in six lines of four, they could just as easily be read by `NumberStats`. This added flexibility is supported by combining string tokenization with `readLine()`. Of course, we have used the `StringTokenizer` class before, but this is our first example of reading numeric data. Note that the `StringTokenizer` constructor receives two arguments. (In our previous examples, only one argument was used — the string to be tokenized.) The second argument is a delimiter string to replace the default delimiter string. We've added the comma (`,`) as a possible delimiter for the tokens. The other allowable delimiters are spaces and tabs. Note that the newline character is also a delimiter, but it is effectively handled (i.e., removed) by `readLine()`.

The process of reading a token, converting it to a double, and placing it in the array (lines 26-27) is straight forward. Once the arrray is filled with data, we just sit back and watch the statistics methods do their job. Lines 33-37 call the methods in turn and print the results.

Graphing Data in an Applet

If you believe a picture is worth a thousand words, then you'll like the next example. The program `GraphData` embeds the data from `euro.dat` in an applet and graphs the progress of the euro's stock market trading value (see Figure 14).

```
 1  import java.awt.*;
 2  import java.applet.*;
 3
 4  public class GraphData extends Applet
 5  {
 6     private static final double[] EURO = {
 7        1.175, 1.176, 1.179, 1.182, 1.188, 1.185, 1.187, 1.188,
 8        1.186, 1.184, 1.180, 1.180, 1.179, 1.180, 1.178, 1.180,
 9        1.180, 1.180, 1.182, 1.181, 1.182, 1.182, 1.183, 1.183
10     };
11     private static final int XSIZE = 500;
12     private static final int YSIZE = 250;
13     private static final int GAP = 50;
14     private static final int WIDTH  = XSIZE - 2 * GAP;
15     private static final int HEIGHT = YSIZE - 2 * GAP;
16
17     public void paint(Graphics g)
18     {
19        // draw rectangle around graphics window
20        g.drawRect(0, 0, XSIZE - 1, YSIZE - 1);
21
22        // set new origin to bottom-left of graph
23        g.translate(GAP, GAP + HEIGHT);
24
25        // draw axes
26        g.drawLine(0, 0, WIDTH, 0);    // x axis
27        g.drawLine(0, 0, 0, -HEIGHT);  // y axis
28
29        // find minimum and maximum values in array
30        double yMin = min(EURO, EURO.length);
31        double yMax = max(EURO, EURO.length);
32
33        // label graph
34        g.setFont(new Font("Helvetica", Font.PLAIN, 14));
35        g.drawString("Euro value (US$) over 24 hr. Jan 3-4, 1999", 75, 16);
36
37        // label y axis with minimum and maximum values
38        g.setFont(new Font("Helvetica", Font.PLAIN, 12));
39        g.drawString("" + yMax, -34, -HEIGHT + 8);
40        g.drawString("" + yMin, -34, 0);
41
42        // find first point of first line to draw
43        int x2 = 0;
44        int y2 = -(int)((EURO[0] - yMin) / (yMax - yMin) * HEIGHT);
45
46        // draw graph (a series of connected lines)
47        for (int i = 1; i < EURO.length; i++)
48        {
49           int x1 = x2;
50           int y1 = y2;
51           x2 = (int)((i / (EURO.length - 1.0)) * WIDTH);
52           y2 = -(int)((EURO[i] - yMin) / (yMax - yMin) * HEIGHT);
53           g.drawLine(x1, y1, x2, y2);
54        }
55     }
56
57     // find the minimum value in a array
58     public static double min(double[] n, int length)
59     {
60        double min = n[0];
61        for (int j = 1; j < length; j++)
62           if (n[j] < min)
```

```
63              min = n[j];
64          return min;
65      }
66
67      // find the maximum value in an array
68      public static double max(double[] n, int length)
69      {
70          double max = n[0];
71          for (int j = 1; j < length; j++)
72              if (n[j] > max)
73                  max = n[j];
74          return max;
75      }
76  }
```

Figure 14. `GraphData.java`

When the applet is run, the graph in Figure 15 appears.

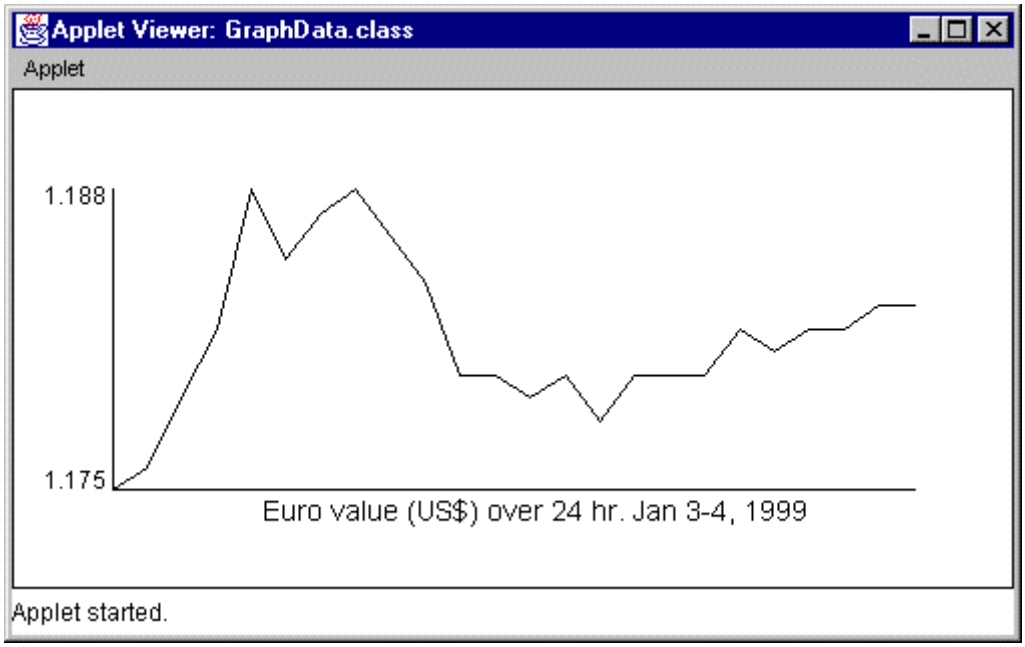


Figure 15. Output of `GraphData` applet

As noted in earlier, applets are distinctly different from Java applications. Applets are small programs embedded in web pages. They are executed through an html document from a browser or using `appletviewer`. There is no "standard input", per se. To get around this, the data in `euro.dat` are embedded in the source code in `GraphData` (see lines 6-10).

Our most important task is "planning". It is one thing to draw a series of connected lines, it is quite another to organize and plan how the graph is laid out — keeping in mind the need to change the data, or to re-size the graph. For this purpose, three defined constants are critical: `XSIZE`, `YSIZE`, and `GAP` (see lines 11-13). `XSIZE` and `YSIZE` are the dimensions of the graphics window, and `GAP` is the space allocated around the inner edge of the window. From these, the additional constants `WIDTH` and `HEIGHT` are defined (lines 14-15) to set the dimensions of the graph. The role of these constants is depicted in Figure 16.

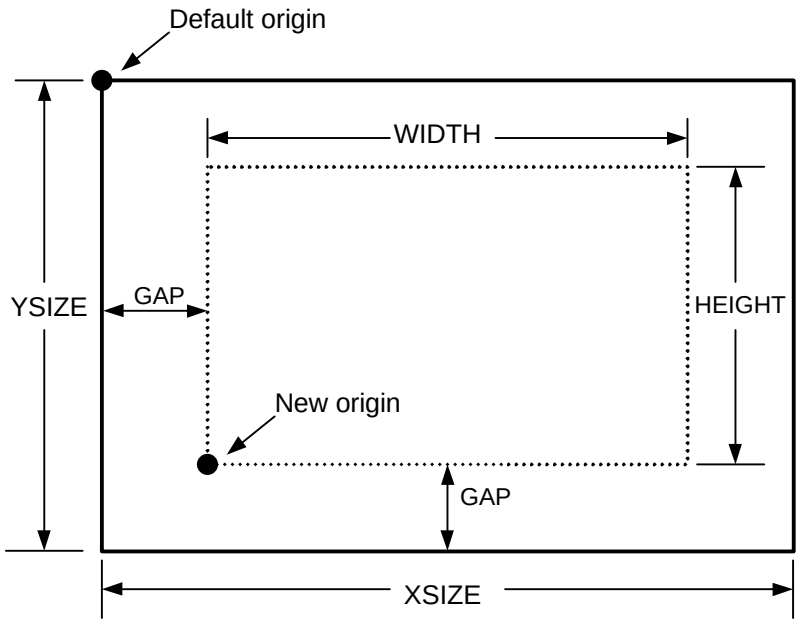


Figure 16. Layout constants used in `GraphData`

The area marked-off by the dotted line in Figure 16 is where the graph is placed. To keep it simple, the gap is the same around all four sides of the graph, thus positioning the graph in the centre of the graphics window.

Before proceeding, we must understand the coordinate system in Java's graphics windows. By default, the origin is point (0, 0) at the top-left corner of the window, as shown in Figure 16. The units are pixels, so all coordinates must be integers. Positive *x* values are to the right of the origin, and positive *y* values are below the origin. This is fine for *x*, but it's not quite what we want for *y*. It is customary to plot data with positive *y* values "above" the origin. Simple "negation" of *y* values will adjust for this. As well, we'd like to work with "our origin", not the graphics window's origin. Once again, there is a simple fix. The `translate()` method in the `Graphics` class allows us to change the origin. Our preferred origin is labeled "New origin" in Figure 16.

Let's walk through the tasks in plotting the euro's trading value. First, a box is drawn around the graphics window (line 20) using the default coordinates. The arguments to the `drawRect()` method are rectangle's *x* coordinate, *y* coordinate, width, and height, respectively. Then, a new origin is set that is well-suited to the chosen layout (line 23). The arguments to the `translate()` method are the *x* and *y* coordinate of the new origin. Subsequent drawing operations are relative to this point. The axes are drawn in lines 26-27. The `drawLine()` method of the `Graphics` class requires four arguments: the *x*-*y* coordinate of the beginning of the line and the *x*-*y* coordinate at the end of the line, respectively. Note that the coordinates are simple and intuitive, given the new origin.

Then, the axes are labeled (lines 34-40). The *y* axis shows only the minimum and maximum values, and these are obtained in lines 30-31 using the `min()` and `max()` methods from the `NumberStats` program presented earlier (see Figure 14).

The data are plotted in a `for` loop that draws a series of lines. Each line begins where the previous line finishes. This explains why `x1` and `y1` are assigned `x2` and `y2`, respectively, in

each new iteration (lines 49-50). With *n* points, it is only necessary to draw *n* - 1 lines; so, the loop control variable, `i`, is initialized at 1 (instead of 0). The coordinates of the first point of the first line are established before the loop begins (lines 43-44).

Although we have a new origin that suits our layout, the raw data must be transformed into *x*-*y* pixel coordinates. The *x* coordinate is implicit: Each point is the value of the euro "one hour after" the previous value. The *y* coordinate is the US dollar value of the euro at that point in time. So, in our graph the *x* axis is "time" and the *y* axis is "US dollars". The *x* coordinate of the first point in the first line is 0 (line 43). Subsequent *x* coordinates are computed in line 51 using the expression

```
(int)((i / (EURO.length - 1.0)) * WIDTH)
```

This expression uses the loop control variable, `i`, to set each new *x* coordinate. By dividing `i` by `EURO.length - 1.0` and multiplying by `WIDTH`, the *x* coordinates are evenly spaced along the *x* axis, given the width of the graph and the number of points plotted.

Transforming the *y* coordinates is tricky. For this, we need the statistics `yMin` and `yMax`, calculated earlier (lines 30-31). The *y* coordinates are computed in line 52 using the following expression:

```
-(int)((EURO[i] - yMin) / (yMax - yMin) * HEIGHT)
```

The effect is to transform euro dollar values such that when `EURO[i] = yMin`, the result is 0, and when `EURO[i] = yMax` the result is `-HEIGHT`. All other values are evenly distributed between these limits. The *y* coordinate of the first point in the first line is calculated in line 44 using the same expression, except with `i` equal to 0. The unary negation operator (`-`) precedes the expression to ensure that positive values are "above" the origin, for reasons noted earlier.

Note in lines 43-54 in `GraphData` that numeric literals are not used (except 0 and 1). There is a great "pay off" in the planning invested in working with pre-defined constants. The `GraphData` applet can be re-sized easily by changing `XSIZE` and `YSIZE`, and the amount of space around the graph is configurable by altering `GAP`. Furthermore, new data can be substituted for the `EURO` array. The new data can have any range and there can be any number of points. The "planning" in `GraphData` is sufficiently flexible that all the pieces fall into place when the program is re-compiled. Of course, there are limits. With `XSIZE` = 250 and `GAP` = 50, we're left with 150 pixels along the *x* axis for the graph. Clearly, we cannot use `GraphData` in its present form to plot an array containing one thousand elements. More sophisticated techniques must be devised.

The other main deficiency in `GraphData` is in the text labels. As seen in lines 34-40, numeric literals are used to "nudge" the text to reasonable positions relative to the axes. Although the effect is pleasing, this approach is bad news if the graph is re-sized or if new text is substituted. Unfortunately, fonts do not scale easily and it is difficult to determine the "pixel" length of a text string; so, a more flexible approach is not possible, given the tools at hand. In the program's present form, changes in the labels require brute-force tuning using a trial and error approach.

Multi-Dimensional Arrays

Multi-dimensional arrays are a useful and powerful extension to simple one-dimensional arrays. In essence, a multi-dimensional array is an "array of arrays". We noted earlier that a `boolean` array could store the status of spaces on a game board. For example, the status of the nine spaces in a Tic Tac Toe game could be represented as

```
boolean[] spaceOccupied = new boolean[9];
```

Since a game board is a two-dimensional space, it may be better to use a data structure that more-closely matches the layout of the board. A two-dimensional array seems particularly applicable in this case:

```
boolean[][] spaceOccupied = new boolean[3][3];
```

A row is identified as an index in the first set of brackets, and a column by as an index in the second set. With the declaration above, the array is initialized with `false` for each element. If the first move of the game were to select and 'X' in the middle of the game board, the array is updated as follows:

```
spacedOccupied[1][1] = true;
```

This may be followed by selecting an 'O' in the bottom-right corner:

```
spaceOccupied[2][2] = true;
```

Figure 17 illustrates the state of the status array and of the game after these two moves.

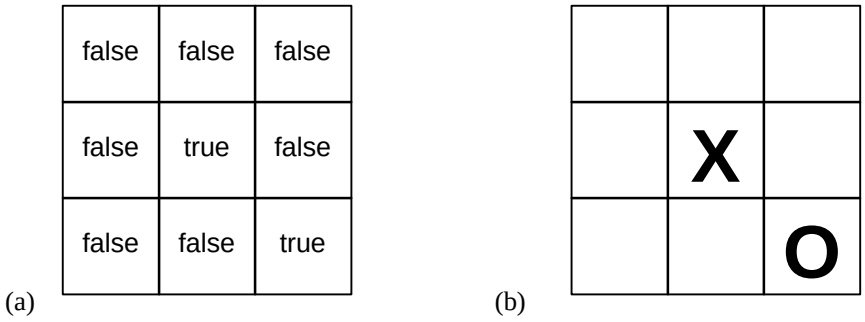


Figure 17. A two-dimensional `boolean` array for the status of a Tic Tac Toe game. Status after two moves for (a) array (b) game

The `spaceOccupied` array could assist in determining if a space selected by the user — identified by its row and column indices — is available:

```
if (spaceOccupied[row][col])
   System.out.println("Sorry, try again!");
```

Of course, multi-dimensional arrays are legitimate data structures for any of Java's primitive data types as well as for objects. As with one-dimensional arrays, an initialization list may be used if the values are known in advance. The syntax is a reasonable extension of that noted earlier for one-dimensional arrays:

```
int[][] abc = {
      {  1,  2,  3 },
      {  4,  5,  6 },
      {  7,  8,  9 },
      { 10, 11, 12 },
   };
```

The array `abc` above is a two-dimensional array with four rows and three columns. Indices may vary from 0 to 3 for the rows and from 0 to 2 for the columns. So, `abc[3][2]` equals 12, but `abc[2][3]` generates an "index out of bounds" exception. The expression `abc.length` equals 4 (the number of rows in the array), whereas the expression `abc[3].length` equals 3 (the number of elements in the last row).

Multi-dimensional arrays needn't be rectangular. The following array declaration is permissible:

```
int[][] triangle = {
      { 1 },
      { 2, 3 },
      { 4, 5, 6 },
      { 7, 8, 9, 10 },
   };
```

The expression `test[n].length` equals the length of the n^{th} row; and so, varies from 1 to 4 as `n` varies from 0 to 3.

Rotating a Graphic Object

A rectangular array is often called a *matrix*. Matrix algebra is an important branch of mathematics, as many common and interesting problems are well expressed and developed using matrices. Let's explore one such example in the field of computer graphics. Figure 18 illustrates a simple house drawn in a graphics window. The house is shown to the right and above the origin. The *y* coordinates are negative as consistent with Java's graphics coordinate system. The house could be drawn using the `drawLine()` method of Java's `Graphics` class, as shown in previous programs. Five calls to `drawLine()` are required, each specifying the end points of one of the lines.

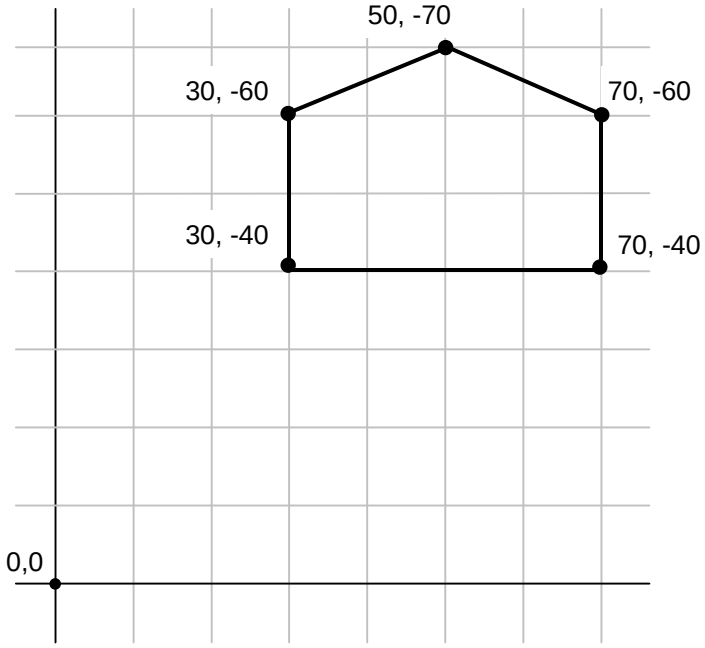


Figure 18. A house displayed in a graphics window

An alternate way to draw the house is to store the points in arrays and use the `drawPolygon()` method:

```
int[] houseX = {  30,  30,  50,  70,  70 };
int[] houseY = { -40, -60, -70, -60, -40 };
g.drawPolygon(houseX, houseY, houseX.length);
```

The `drawPolygon()` method requires three arguments: an array of *x* points, an array of *y* points, and the number of points. So far, so good. If we want to transform our basic house in

some simple yet consistent manner, then we must confront a mathematical problem that is well-suited to matrix algebra, and, hence, multi-dimensional arrays. In computer graphics, there are three basic transformations collectively known as *affine transformations*:

Translating	moving an object left, right, up, or down
Scaling	enlarging or shrinking an object
Rotation	rotating an object about a point

The task of translating, scaling, or rotating our simple graphics house requires transforming each point into a new point such that the new set of points is true to the original set, except transformed in the desired manner. For translating, we simply add a displacement to each *x* and *y* coordinate:

$$x' = x + d_x \qquad y' = y + d_y \tag{1}$$

For scaling, we simply multiply each *x* and *y* coordinate by a scaling factor:

$$x' = x \cdot s_x \qquad y' = y \cdot s_y \tag{2}$$

For rotating, we transform each point as follows:

$$x' = x \cdot \cos q - y \cdot \sin q \qquad y' = x \cdot \sin q + y \cdot \cos q \tag{3}$$

where q is the angle of rotation in degrees. Rotating is a bit tricky, so let's explain with a figure. Figure 19 shows a point *x*, *y*, and the same point, *x'*, *y'*, rotated about the origin by q degrees. Since rotation is about the origin, the distance from the origin to each point is the same. This is shown as *z* in the figure.

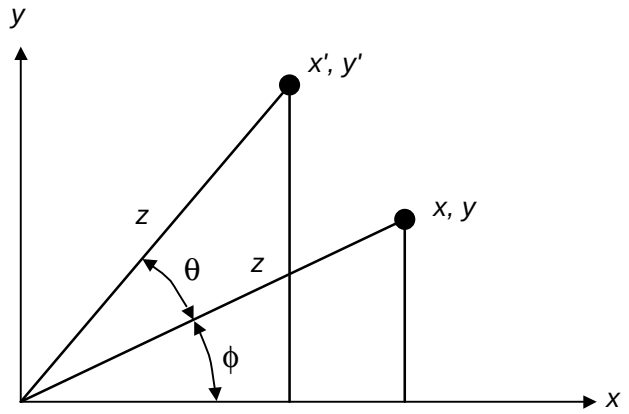


Figure 19. Rotating a point about the origin

Our task is to calculate *x'* and *y'* in terms of *x*, *y*, and the angle of rotation, q. From basic trigonometry, we know that

$$x = z \cdot \cos f \tag{4}$$

$$y = z \cdot \sin f \tag{5}$$

In terms of the new point, *x'*, *y'*, we have

$$x' = z \cdot \cos(\theta + \phi) = z \cdot \cos\phi \cdot \cos\theta - z \cdot \sin\phi \cdot \sin\theta \tag{6}$$

$$y' = z \cdot \sin(\theta + \phi) = z \cdot \cos\phi \cdot \sin\theta + z \cdot \sin\phi \cdot \cos\theta \tag{7}$$

Substituting equations 4 and 5 into equations 6 and 7 yields the equations for *x'* and *y'* (equation 3).

So, what does the above discussion have to do with multi-dimensional arrays or matrix algebra? A lot. As it turns out, the three affine transformations, and some variations on these, can all be implemented using the same operation — multiplying a vector by a transformation matrix. The arithmetic is simple, but as you may unfamiliar with vector and matrix algebra, a brief review is in order.

A vector is another term for an array, and a matrix is another term for a two-dimensional array.[2] A vector multiplied by another vector yields a single value as a result, for example

$$\begin{pmatrix} 6 & 5 & 4 \end{pmatrix} \cdot \begin{pmatrix} 9 \\ 8 \\ 7 \end{pmatrix} = 6 \times 9 + 5 \times 8 + 4 \times 7 = 54 + 40 + 28 = 122 \tag{8}$$

A vector multiplied by matrix yields a vector result, for example

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \cdot \begin{pmatrix} 4 & 5 & 6 \\ 3 & 2 & 1 \\ 7 & 6 & 5 \end{pmatrix} = \begin{pmatrix} 1\times4+2\times5+3\times6 \\ 1\times3+2\times2+3\times1 \\ 1\times7+2\times6+3\times5 \end{pmatrix} = \begin{pmatrix} 4+10+18 \\ 3+4+3 \\ 7+12+15 \end{pmatrix} = \begin{pmatrix} 32 \\ 10 \\ 34 \end{pmatrix} \tag{9}$$

We'll put the operation above to use in our next Java program, so have another look if you're not convinced of the arithmetic. For our purpose, the vector represents a point to transform:

$$P = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{10}$$

The matrix used to transform this point will differ depending on whether we are interested in translating (T), scaling (S), or rotating (R):

$$T = \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix} \qquad S = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad R = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{11}$$

So, if we are interested in translating an object, we take each point, *P*, and transform it into *P'* using

$$P' = P \cdot T \tag{12}$$

---

[2] The term "vector", as it is used here, is unrelated to Java's `Vector` class, discussed in the next section.

If we want to scale our object, each point, *P*, is transformed into *P′* using

$$P' = P \cdot S \tag{13}$$

and for rotating, each point, *P*, is transformed into *P′* using

$$P' = P \cdot R \tag{14}$$

The third value in the point vector is of no use to us, so it is discarded after the transformation.

As an example, let's convince ourselves that equation 12 performs translation:

$$P \cdot T = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} x \times 1 + y \times 0 + 1 \times d_x \\ x \times 0 + y \times 1 + 1 \times d_y \\ x \times 0 + y \times 0 + 1 \times 1 \end{pmatrix} = \begin{pmatrix} x + d_x \\ y + d_y \\ 1 \end{pmatrix} = P' \tag{15}$$

The result above shows the translated point with new coordinates $x + d_x$ and $y + d_y$. This is the effect of translating, as expressed in equation 1 at the beginning of this section.

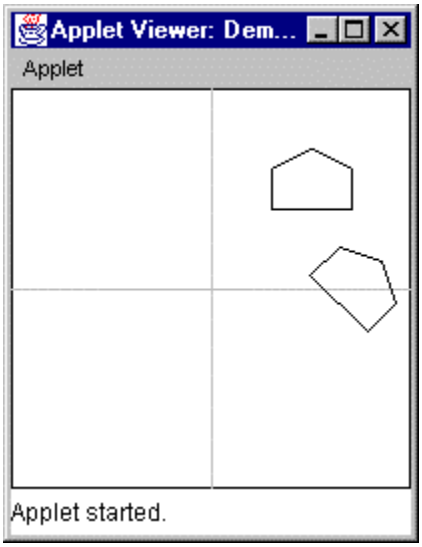
The transformations above are powerful tools for computer graphics. Let's demonstrate this using an example program. The program `DemoRotate` is a Java applet that draws the simple house shown earlier and then draws it again rotated 45° about the origin (see Figure 20).

```
 1  import java.awt.*;
 2  import java.applet.*;
 3
 4  public class DemoRotate extends Applet
 5  {
 6     private static final int SIZE = 200;
 7     private static final double ROTATE = 45.0;  // degrees cw
 8
 9     public void paint(Graphics g)
10     {
11        // points for the house polygon
12        int[] houseX = {  30,  30,  50,  70,  70 };
13        int[] houseY = { -40, -60, -70, -60, -40 };
14
15        // compute rotation angle in radians
16        double theta = Math.toRadians(ROTATE);
17
18        // transformation matrix for rotation
19        double[][] r = {
20           { Math.cos(theta), -Math.sin(theta), 0.0 },
21           { Math.sin(theta),  Math.cos(theta), 0.0 },
22           {             0.0,              0.0, 1.0 },
23        };
24
25        // draw a rectangle around the graphics window
26        g.drawRect(0, 0, SIZE - 1, SIZE - 1);
27
28        // set a new origin in the middle of the graphics window
29        g.translate(SIZE / 2, SIZE / 2);
30
31        // draw the x axis and y axis (in light gray)
32        g.setColor(Color.lightGray);
33        g.drawLine(0, -(SIZE - 1), 0, +(SIZE - 1));
34        g.drawLine(-(SIZE - 1), 0, +(SIZE - 1), 0);
35
36        // draw outline of the original house (in black)
37        g.setColor(Color.black);
38        g.drawPolygon(houseX, houseY, houseX.length);
39
40        // transform points (use 'rotate' matrix)
41        double[] p = new double[3];
42        for (int i = 0; i < houseX.length; i++)
43        {
44           p[0] = houseX[i];
45           p[1] = houseY[i];
46           p[2] = 1.0;
47           p = vmProduct(p, r);
48           houseX[i] = (int)Math.round(p[0]);
49           houseY[i] = (int)Math.round(p[1]);
50        }
51
52        // draw the transformed house
53        g.drawPolygon(houseX, houseY, houseX.length);
54     }
55
56     // calculate product of vector 'v' and matrix 'm' (return a new vector)
57     public static double[] vmProduct(double[] v, double[][] m)
58     {
59        double[] temp = new double[v.length];
60        for (int i = 0; i < v.length; i++)
61           temp[i] = v[0] * m[i][0] + v[1] * m[i][1] + v[2] * m[i][2];
62        return temp;
```

```
63      }
64  }
```

Figure 20. `DemoRotate.java`

This program generates the following output:



The techniques to draw the original house were covered earlier. The interesting part in this program is the implementation of the affine transformation for rotation using matrix algebra. The transformation matrix is initialized in lines 19-23 as a two-dimensional `double` array named `r` (for "rotate"). The values in the array are consistent with the transformation matrix for rotation presented earlier (see equation 11). The amount of rotation is declared in the `final` constant `ROTATE` in line 7 as 45°. Since Java's `sin()` and `cos()` methods expect an angle in radians, the `toRadians()` method (line 16) transforms the angle before it is passed on to these methods.

After the original object is drawn, the affine transformation is applied to the array of points (lines 41-50). In turn, each point is read from the original array `houseX` and `houseY` and assigned to a temporary point array named `p` (lines 44-46). This point and the transformation matrix `r` are passed as arguments to the `vmProduct()` method in line 47. The `vmProduct()` method multiplies the vector (`p`) by the matrix (`r`) and returns a new vector in `p`. The first two values in `p` are placed back into the `houseX` and `houseY` array as the newly transformed points (lines 48-49).

The vector-matrix multiplication in the `vmProduct()` method is a straight forward implementation of the arithmetic shown earlier in equation 9. After the transformation the rotated house is drawn (line 53).

See Chapter 5 of Foley et al's *Computer graphics: Principles and practice* for more details on 2D transformations for computer graphics.

Vectors

The size of an array is set when the array is declared, and the array cannot grow or shrink thereafter. This presents a problem when data are read from an external source and the number of data items is not known in advance. We used a work-around earlier by declaring a "larger than necessary" array and then maintaining a variable to hold the "actual" number of elements initialized. Obviously, this is wasteful. It is also possible that someday more data items are read than anticipated. In this case the program simply won't work unless the source code is updated and the program is recompiled. These problems are averted using a *dynamic array*.

Java's `Vector` class implements a dynamic array of objects. Like an array, it contains elements accessible using a simple integer index. However, the size of a `Vector` object can grow or shrink to accommodate adding and removing items on an as-needed basis. Of course, there is behind-the-scenes storage management taking place, but this is transparent to us.

By way of introduction, we'll digress briefly with an example to convince you of the need for dynamic arrays. Let's revisit two programs shown earlier: `RandomGen` and `NumberStats`. The `RandomGen` program outputs random numbers to the standard output stream, and the `NumberStats` program analyses data read from the standard input stream. That's great because we can connect the two programs using a pipe. (Pipes were presented earlier.) A sample dialogue follows:

```
PROMPT>java RandomGen 100 400.0 600.0 | java NumberStats
N = 100
Minimum = 402.17233767914325
Maximum = 599.6718556474029
Mean = 506.783572075286
Standard deviation = 55.920094678999526
```

In the first part of the command line, the `RandomGen` program generates 100 random floating-point numbers ranging from 400 to 600. Normally these values are sent to the host system's CRT display; however, the pipe symbol (`|`) effectively "catches" the output and sends it as input to the `NumberStats` program (via the `java` interpreter). The output is not surprising given our understanding of these programs. However, if we are interested in gathering statistics on, say, 150 numbers, then a problem is lurking. Here is another sample dialogue:

```
PROMPT>java RandomGen 150 400.0 600.0 | java NumberStats
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
       at NumberStats.main(Compiled Code)
```

The problem lies in the `NumberStats` program. If you recall, the `numbers` array was declared with a default size of 100 (see line 13 `NumberStats.java`). As soon as the 101st element arrived via the pipe, the program crashed with an "array index out of bounds" exception. The most obvious fix is to edit the source code and recompile with yet another "larger than necessary" array size. But this is a nuisance.[1]

[1] It is possible to create pseudo-dynamic arrays as follows. When the capacity of the original array is reached, a new, larger array is declared and the elements from the old array are copied into the new array. The old array is discarded. If the new array also reaches its capacity, the process is repeated, and so on.

Dynamic Arrays of Primitive Data Types

An alternative approach, which we will explore now, is to re-work the `NumberStats` program using the `Vector` class, instead of arrays. A key difference between `Vector` objects and arrays is that the former only hold objects. To store primitive data types, like integers, characters, or floating-point numbers, the values must be encoded as wrapper-class objects. (Java's wrapper classes were presented earlier.)

An object of the `Vector` class is instantiated like an object of any other class — using a constructor method. Once instantiated, elements are added and retrieved using instance methods. The `Vector` class methods are summarized in Table 1.

Table 1. Methods of the `Vector` Class

Method	Description
Constructor	
`Vector()`	constructs an empty vector so that its internal data array has size 10 and its standard capacity increment is zero; returns a reference to the new object
Instance Methods	
`addElement(Object obj)`	adds the specified element to the end of this vector, increasing its size by one; returns a `void`
`add(int index, Object o)`	Adds the specified element at the spcified position to the this vector; returns a `void`
`add(Object o)`	Adds the specified element to the end of this vector; returns a `boolean`
`elementAt(int index)`	returns an `Object` representing the element at the specified index (same as `get()`)
`get(int index)`	returns an `Object` representing the element at the specified index. (same as `elementAt()`)
`removeElementAt(int index)`	deletes the element at the specified index; returns a void
`size()`	returns an `int` equal to the number of elements in this vector

Let's assume we need a dynamic array (a `Vector`) of integers. The follow statement instantiates a `Vector` object named `samples`:

```
Vector samples = new Vector();
```

To add an element to the vector, we use the `addElement()` method. However, as shown in Table 1, the argument must be an `Object`. Since all Java classes are subclasses of the `Object` class, we can use any object reference as an argument. We cannot, however, use a primitive data type. If we wish to add an `int` to `samples`, it first must be "wrapped" into an `Integer` object. If, for example, the value of the integer is 5, it is instantiated as an `Integer` object using

```
Integer x = new Integer(5);
```

and then added to `samples` using the `addElement()` instance method:

```
samples.addElement(x);
```

Of course, we have no use for the variable `x`, so we can combine the two steps above:

```
samples.addElement(new Integer(5));
```

Note that the `Integer` constructor also accepts a string representation of an integer as an argument. For example

```
samples.addElement(new Integer("5"));
```

Additional elements are added to `samples` in a similar manner. The best part is this: We need not concern ourselves with the capacity of `samples`. Extra space is allocated on an as-needed basis.

To retrieve an element from `samples`, we use the `elementAt()` instance method. The argument is a simple integer index specifying the location of the element we wish to retrieve. Once again, we have a minor problem because `samples` is a `Vector` object containing a dynamic array of `Object` objects. We cannot retrieve an element and assign it to an `int` variable:

```
int y = samples.elementAt(i);  // Wrong!
```

The `elementAt()` method returns an `Object` (actually, a reference to an `Object`). We can assign it to an `Integer` wrapper class object, but only if the `Object` reference is cast to an `Integer` reference:[2]

```
Integer temp = (Integer)samples.elementAt(i);
```

That's half the story. What we really want is to retrieve an element and assign it to an `int`. So, we follow the statement above with

```
int y = temp.intValue();
```

We can avoid instantiating the `Integer` object `temp` by combining the two preceding statements:

```
int x = ((Integer)samples.elementAt(i)).intValue();
```

The extra parentheses are needed to ensure the operations occur in the correct order.

At any time, we can retrieve the number of elements stored in the vector using the `size()` method:

```
for (int i = 0; i < samples.size(); i++)
...
```

So, unlike our experience with arrays, we do not need a separate variable representing the number of initialized entries.

---

[2] We did not cast the `Integer` reference to a `Object` reference when using `addElement()` because of the inheritance relationship between the `Integer` (subclass) and `Object` (superclass) classes: an `Integer` object is an `Object` object (always!). However, we must cast the `Object` reference to an `Integer` reference when going the other way using `elementAt()` because an `Object` object is not necessarily a `Integer` object.

Let's put these pieces together in a new version of the `NumberStats` program that uses dynamic arrays. The program `NumbersStats2` inputs any number of floating point numbers from the standard input and outputs some simple statistics on these numbers (see Figure 1).

```
 1  import java.io.*;
 2  import java.util.*;
 3
 4  public class NumberStats2
 5  {
 6     public static void main(String[] args) throws IOException
 7     {
 8        BufferedReader stdin =
 9           new BufferedReader(new InputStreamReader(System.in), 1);
10
11        // declare 'v' as a dynamic array (a Vector object)
12        Vector v = new Vector();
13
14        // input data and put into dynamic array
15        String line;
16        while ((line = stdin.readLine()) != null)
17        {
18           // prepare to tokenize line
19           StringTokenizer st = new StringTokenizer(line, " ,\t");
20
21           // process tokens in line
22           while (st.hasMoreTokens())
23           {
24              String s = st.nextToken();  // get a token
25              Double d = new Double(s);   // convert and wrap as Double
26              v.addElement(d);            // add to dynamic array
27           }
28        }
29
30        // declare a double array with exactly the size needed
31        double[] numbers = new double[v.size()];
32
33        // copy and convert elements of Vector object into double array
34        for (int i = 0; i < v.size(); i++)
35           numbers[i] = ((Double)v.elementAt(i)).doubleValue();
36
37        // output some statistics on the data
38        System.out.println("N = " + numbers.length);
39        System.out.println("Minimum = " + min(numbers));
40        System.out.println("Maximum = " + max(numbers));
41        System.out.println("Mean = " + mean(numbers));
42        System.out.println("Standard deviation = " + sd(numbers));
43     }
44
45     // find the minimum value in an array
46     public static double min(double[] n)
47     {
48        double min = n[0];
49        for (int j = 1; j < n.length; j++)
50           if (n[j] < min)
51              min = n[j];
52        return min;
53     }
54
55     // find the maximum value in a array
56     public static double max(double[] n)
57     {
58        double max = n[0];
59        for (int j = 1; j < n.length; j++)
60           if (n[j] > max)
61              max = n[j];
62        return max;
```

```
63     }
64
65     // calculate the mean of the values in an array
66     public static double mean(double n[])
67     {
68        double mean = 0.0;
69        for (int j = 0; j < n.length; j++)
70           mean += n[j];
71        return mean / n.length;
72     }
73
74     // calculate the standard deviation of values in an array
75     public static double sd(double[] n)
76     {
77        double m = mean(n);
78        double t = 0.0;
79        for (int j = 0; j < n.length; j++)
80           t += (m - n[j]) * (m - n[j]);
81        return Math.sqrt(t / (n.length - 1.0));
82     }
83  }
```

Figure 1. `NumberStats2.java`

Now we can do what our original version of `NumberStats` could not:

```
PROMPT>java RandomGen 150 400.0 600.0 | java NumberStats2
N = 150
Minimum = 400.4034372537817
Maximum = 598.4001802690623
Mean = 498.7361024890425
Standard deviation = 55.27611100869566
```

A `Vector` object named `v` is declared in line 12. Values are read from the standard input, as before, except now they are placed in the dynamic array `v` instead of in an over-sized `double` array. This involves converting the inputted `String` to a `Double` object (line 25) and then inserting it into the dynamic array using the `addElement()` method of the `Vector` class (line 26). Once all the elements are read, we can declare a `double` array with precisely the space need. This is done in line 31 using the `size()` method of the `Vector` class to specified the size of a `double` array named `numbers`.

For serious number crunching, it is much more efficient to work with a `double` array than with a vector containing a dynamic array of `Double` objects. So, the elements of the `Vector` object are retrieved, converted, and copied into the double array `numbers` (lines 34-35). The expression

```
((Double)v.elementAt(i)).doubleValue()
```

performs the convoluted task of retrieving the element at position `i`, casting it to a `Double` object, and converting it to a `double`. The result is assigned to position `i` of the `double` array `numbers`.

At this point, it appears we are in basically the same position as in the original program, `NumberStats`. We have a `double` array of values, and we wish to calculate some simple statistics on the values. In fact, we are in a much better position. Because `numbers` is precisely the correct size, we do not need a separate variable for the "actual" number of initialized elements in the array: All elements are initialized! We can now work with the `length` field of the array

— a much more elegant approach. All the statistics methods are modified slightly with this improvement. It is no longer necessary to pass the "size" of the array to the statistics methods. We simply pass the array reference and the `for` loops are setup using the `length` field of the array.

Dynamic Arrays of Objects

In the preceding example, the `Vector` class came to the rescue. We inputted an unspecified number of floating point values and created a `double` array with precisely the space needed. A dynamic array of `Vector` objects was used as an intermediate step, between inputting the values and creating the `double` array. However, the values had to be "wrapped" and "unwrapped" because the `Vector` class only works with objects of the `Object` class. This was inconvenient, but it was well worth the extra effort.

If the data of interest are objects (as opposed to primitive data types), then the `Vector` class is simple and natural to work with. This is illustrated in the following example working with `String` objects. The program `MedalWinners` uses a `Vector` object to hold the names of the medal winners in an Olympic event (see Figure 2).

```
 1  import java.util.*;
 2
 3  public class MedalWinners
 4  {
 5     private static final String[] MEDAL = { "GOLD", "SILVER", "BRONZE" };
 6
 7     public static void main(String[] args)
 8     {
 9        // declare 'topThree' as a Vector object (a dynamic array)
10        Vector topThree = new Vector();
11
12        // add entries (top three finishers in men's 100 meter event)
13        topThree.addElement("Ben Johnson (CAN), 9.79 s");
14        topThree.addElement("Carl Lewis (USA), 9.92 s");
15        topThree.addElement("Linford Christie (GBR), 9.97 s");
16
17        // print results
18        System.out.println("1988 Seoul Olympics, Men's 100 M");
19        for (int i = 0; i < topThree.size(); i++)
20           System.out.println(MEDAL[i] + ":\t" + topThree.elementAt(i));
21
22        // remove Ben Johnson (disqualified)
23        topThree.removeElementAt(0);
24
25        // add new bronze medal winner
26        topThree.addElement("Calvin Smith (USA), 9.99 s");
27
28        // print revised results
29        System.out.println();
30        System.out.println("1988 Seoul Olympics, Men's 100 M (revised)");
31        for (int i = 0; i < topThree.size(); i++)
32           System.out.println(MEDAL[i] + ":\t" + topThree.elementAt(i));
33     }
34  }
```

Figure 2. `MedalWinners.java`

This program generates the following output:

```
1988 Seoul Olympics, Men's 100 M
GOLD:    Ben Johnson (CAN), 9.83 s
SILVER:  Carl Lewis (USA), 9.92 s
BRONZE:  Linford Christie (GBR), 9.97 s

1988 Seoul Olympics, Men's 100 M (revised)
GOLD:    Carl Lewis (USA), 9.92 s
SILVER:  Linford Christie (GBR), 9.97 s
BRONZE:  Calvin Smith (USA), 9.99 s
```

And the rest is history!

A `Vector` class object — a dynamic array — named `topThree` is declared in line 10. Three `String` objects are added to the array in lines 13-15. The `addElement()` method requires an `Object` as an argument, however an object of any class is a valid argument because all classes are subclasses of `Object`. As each element is added, the size of the dynamic array increases by one. The three elements are printed in lines 19-20, and you can see the effect in the first part of the program's output above.

In the print statement in line 18, the following expression appears:

```
topThree.elementAt(i)
```

The `elementAt()` method returns an `Object` reference; however, the `Object` is automatically converted to a string because of the concatenation operator. Be aware, however, that the following attempt to retrieve the gold-medal winner would generate a compile error:

```
String goldMedalWinner = topThree.elementAt(0);  // Wrong!
```

The `Object` returned by the `elementAt()` method cannot be assigned to a `String` object unless it is cast:

```
String goldMedalWinner = (String)topThree.element(0);  // OK!
```

The statement above is good, clean Java code. Unfortunately, Mr. Johnson was not as clean. His use of metabolic steroids necessitated his removal from the Olympic record book. And so, too, we remove his entry from the `topThree` array in line 23 using the `removeElementAt()` method of the `Vector` class. The size of the array is automatically reduced by one. The previous Silver and Bronze medal winners are "bumped up" by one, and a new Bronze medal winner is added at the end of the array in line 26. The size of the array is then automatically increased by one. The final standings are outputted and appear in the last part of the program's output.

Key Points – Arrays and Vectors

We have explored a variety of new ways to work with data — namely through array data structures or using Java's `Vector` class. Here are the key points we have learned:

- Arrays are useful data structures to store a collection of data of the same type.
- Java supports arrays of objects (actually object references) as well as arrays of primitive data types.
- Elements in an array are accessed through an integer index. (For example, if `inventory` is an array and `i` is an integer, `inventory[i]` is the i^{th} element in the array.)
- For each array a public data field named `length` holds the length of the array. (For example, if `gizmo` is an array, `gizmo.length` is the length of the array.)
- A common error in using arrays is attempting to access an element that does not exist. This error is not caught by the compiler; rather, it causes an "array index out of bounds" run-time error.
- Valid array indices range from 0 to "one less than" the length of the array. (For example, if `key` is an array the last element in the array is `key[key.length - 1]`.)
- An array can be initialized when it is declared using an *initialization list*. (For example, `String[] medals = { "GOLD", "SILVER", "BRONZE" };` .)
- An array is an object, however its implementation as an object is through the compiler and the run-time interpreter.
- Java supports multi-dimensional arrays, for example, to represent matrices.
- An array is a *static* data structure. Once an array is declared, its size cannot change.
- Java supports *dynamic* arrays through its `Vector` class.
- A `Vector` class object is a dynamic array of objects. Its size increases and decreases automatically as elements are removed or added.
- Objects stored in or retrieved from a `Vector` object are of the `Object` class.
- If an object retrieved from a dynamic array (a `Vector` object) must be cast before it can be assigned to an object variable (unless the object variable is of the `Object` class.)