

## 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<sup>th</sup> 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.

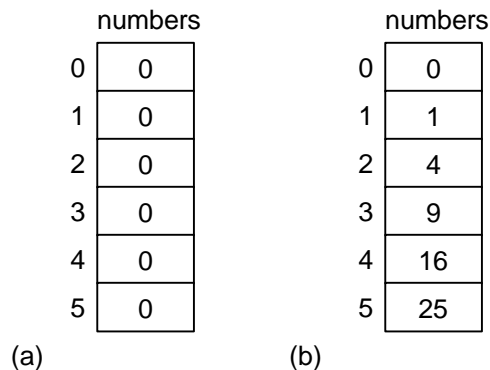


Figure 2. Memory assignments in `DemoArray` (a) after line 5 (b) after line 11

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 progresses, 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 progresses, 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.

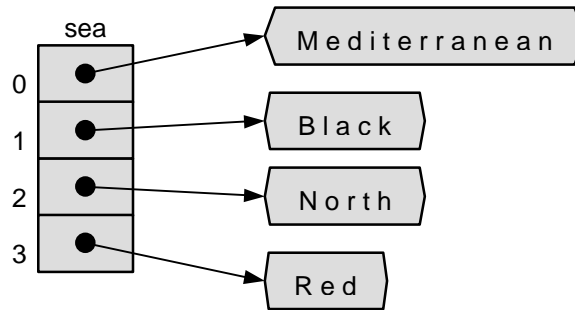


Figure 4. Memory assignments for an array of String objects

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 4<sup>th</sup> 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.

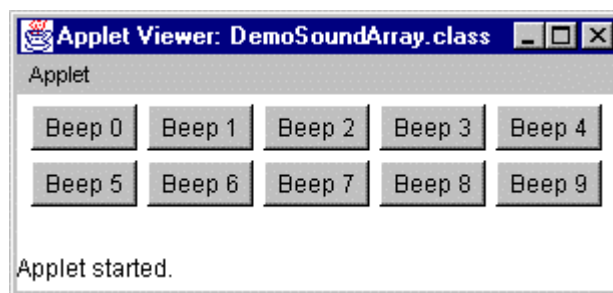


Figure 7. Graphic window from DemoSoundArray applet

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.<sup>1</sup> 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).

---

<sup>1</sup> 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()`:

<code>min()</code>	find the minimum value in an array (lines 41-48)
<code>max()</code>	find the maximum value in an array (lines 51-58)
<code>mean()</code>	calculate the mean of the values in an array (lines 61-67)
<code>sd()</code>	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:

$$\text{mean} = \bar{X} = \frac{\sum x_i}{n}$$

$$\text{sd} = \sqrt{\frac{\sum (\bar{X} - x_i)^2}{n - 1}}$$

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 calculates 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 is 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 to the formal use of the same term in the Java language.) *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 `NumberStats` that is not particularly elegant. 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 expression `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 imperative that `size` is used, not `numbers.length`. In fact, all the statistical methods were defined to receive 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 array 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.

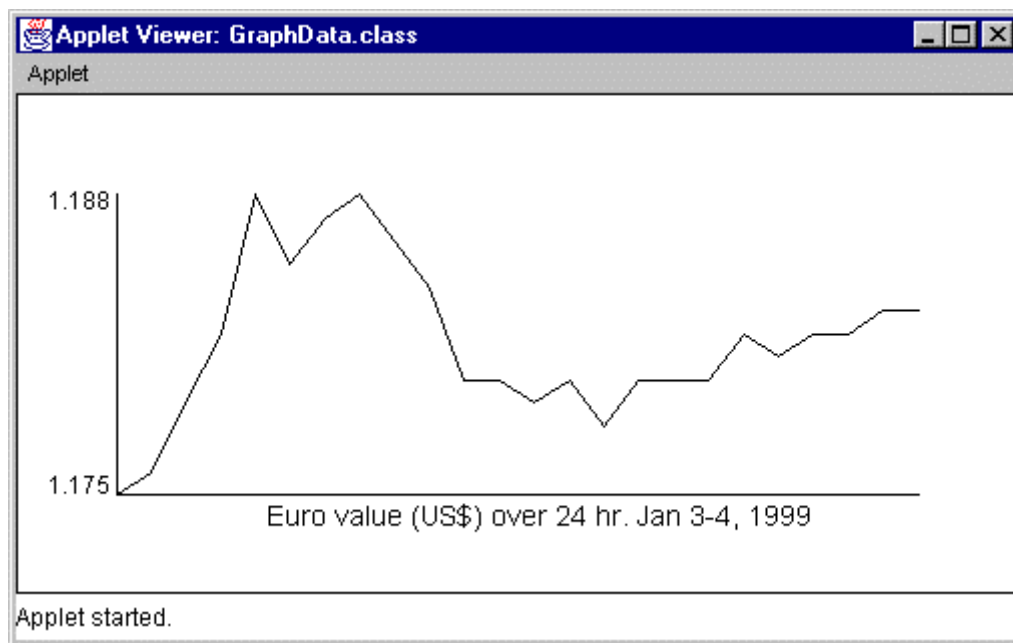


Figure 15. Output of GraphData applet

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.

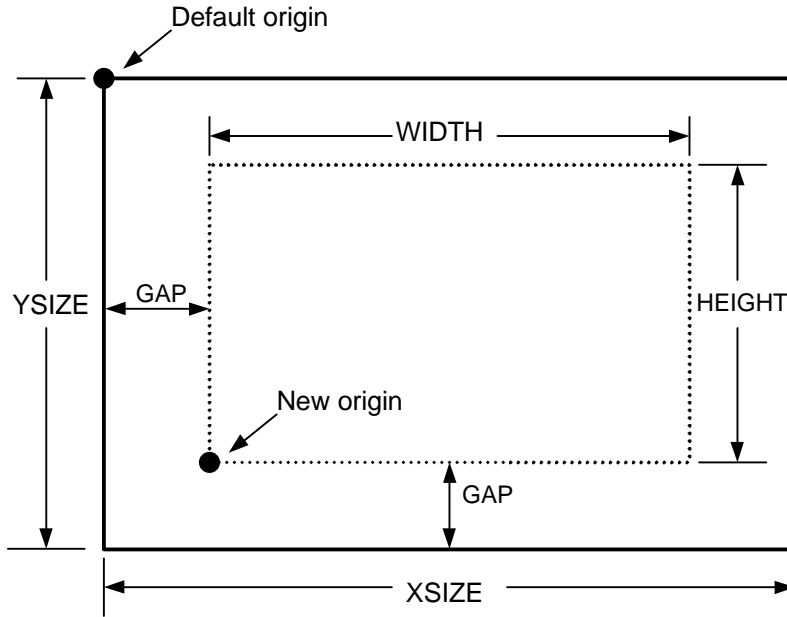


Figure 16. Layout constants used in GraphData

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:

```
spaceOccupied[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.

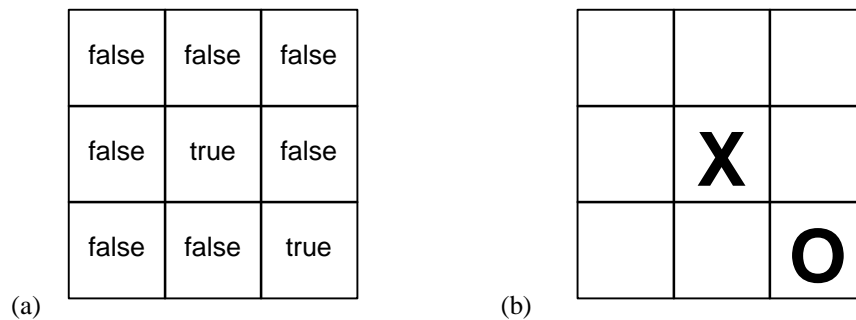


Figure 17. A two-dimensional `boolean` array for the status of a Tic Tac Toe game. Status after two moves for (a) array (b) game

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  $n^{\text{th}}$  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.

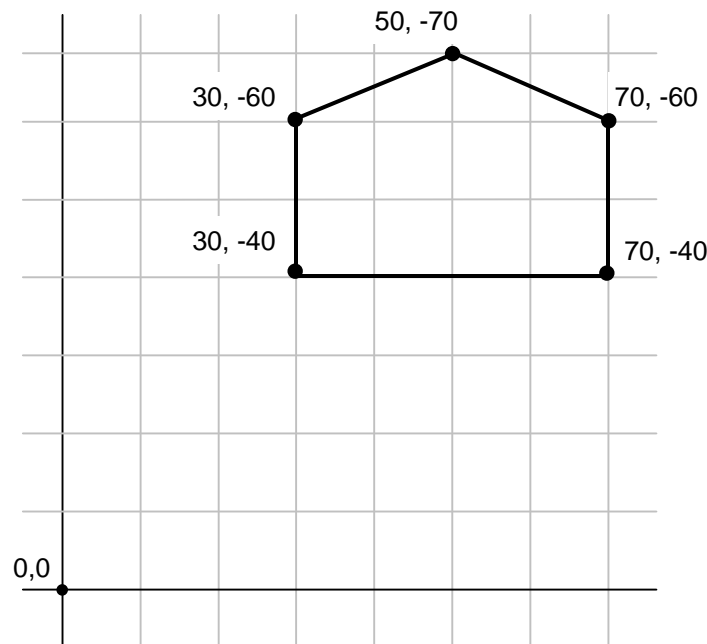


Figure 18. A house displayed in a graphics window

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:

$$x' = x + d_x \qquad y' = y + d_y \qquad (1)$$

For scaling, we simply multiply each  $x$  and  $y$  coordinate by a scaling factor:

$$x' = x \cdot s_x \qquad y' = y \cdot s_y \qquad (2)$$

For rotating, we transform each point as follows:

$$x' = x \cdot \cos \mathbf{q} - y \cdot \sin \mathbf{q} \qquad y' = x \cdot \sin \mathbf{q} + y \cdot \cos \mathbf{q} \qquad (3)$$

where  $\mathbf{q}$  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  $\mathbf{q}$  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.

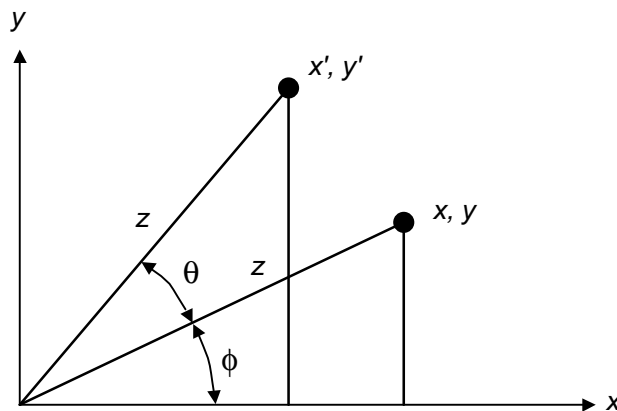


Figure 19. Rotating a point about the origin

Our task is to calculate  $x'$  and  $y'$  in terms of  $x, y$ , and the angle of rotation,  $\mathbf{q}$ . From basic trigonometry, we know that

$$x = z \cdot \cos \mathbf{f} \qquad (4)$$

$$y = z \cdot \sin \mathbf{f} \qquad (5)$$

In terms of the new point,  $x', y'$ , we have

$$x' = z \cdot \cos(\mathbf{q} + \mathbf{f}) = z \cdot \cos \mathbf{f} \cdot \cos \mathbf{q} - z \cdot \sin \mathbf{f} \cdot \sin \mathbf{q} \quad (6)$$

$$y' = z \cdot \sin(\mathbf{q} + \mathbf{f}) = z \cdot \cos \mathbf{f} \cdot \sin \mathbf{q} + z \cdot \sin \mathbf{f} \cdot \cos \mathbf{q} \quad (7)$$

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.<sup>2</sup> A vector multiplied by another vector yields a single value as a result, for example

$$(6 \ 5 \ 4) \cdot \begin{pmatrix} 9 \\ 8 \\ 7 \end{pmatrix} = 6 \times 9 + 5 \times 8 + 4 \times 7 = 54 + 40 + 28 = 122 \quad (8)$$

A vector multiplied by matrix yields a vector result, for example

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \cdot \begin{pmatrix} 4 & 5 & 6 \\ 3 & 2 & 1 \\ 7 & 6 & 5 \end{pmatrix} = \begin{pmatrix} 1 \times 4 + 2 \times 5 + 3 \times 6 \\ 1 \times 3 + 2 \times 2 + 3 \times 1 \\ 1 \times 7 + 2 \times 6 + 3 \times 5 \end{pmatrix} = \begin{pmatrix} 4 + 10 + 18 \\ 3 + 4 + 3 \\ 7 + 12 + 15 \end{pmatrix} = \begin{pmatrix} 32 \\ 10 \\ 34 \end{pmatrix} \quad (9)$$

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:

$$P = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad (10)$$

The matrix used to transform this point will differ depending on whether we are interested in translating (T), scaling (S), or rotating (R):

$$T = \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix} \quad S = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad R = \begin{pmatrix} \cos \mathbf{q} & -\sin \mathbf{q} & 0 \\ \sin \mathbf{q} & \cos \mathbf{q} & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (11)$$

So, if we are interested in translating an object, we take each point,  $P$ , and transform it into  $P'$  using

$$P' = P \cdot T \quad (12)$$

---

<sup>2</sup> 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  $45^\circ$  about the origin (see Figure 20).

```

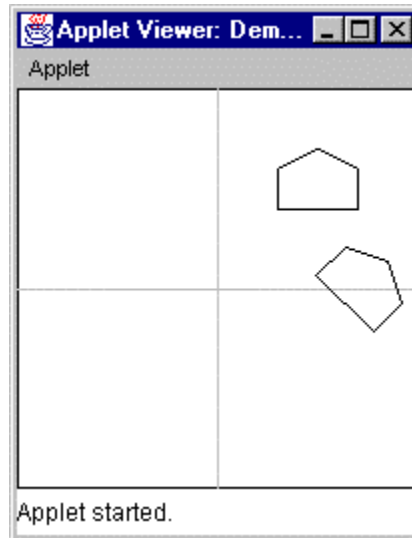
1  import java.awt.*;
2  import java.applet.*;
3
4  public class DemoRotate extends Applet
5  {
6      private static final int SIZE = 200;
7      private static final double ROTATE = 45.0; // degrees cw
8
9      public void paint(Graphics g)
10     {
11         // points for the house polygon
12         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  $45^\circ$ . 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.