Chapter 21

Arrays as Objects

An array is simply an object reference (see Section 20.1). As such, it can be used like any object reference; an array can be

- passed as an actual parameter to a method and
- returned from a method as a return value.

In the previous chapter we saw how access and work with the elements within an array. In this chapter we investigate how manipulate an array itself as a whole.

21.1 Array Parameters

Array parameters. An array is specified as a formal parameter in a method definition with the same syntax as a variable declaration:

```java
// Method f accepts two arrays and a Boolean
public int f(int[] numList, double[] vector, flag) {
    /* . . . Details omitted . . . */
}
```

Calling code simply passes the array as it would any variable:

```java
int[] list = new int[100];
double[] v = new double[3];
/* Initialize list and v (details omitted) */
// . . . then call f()
list[3] = f(list, v, true);
```

The following method sums the elements of an integer array:

```java
public static int sumUp(int[] a) {
    int sum = 0;
```
for ( int value : a ) {
    sum += value;
}
return sum;

Since the method accepts any integer array with no additional information about its size, the number of elements in the array must be determined from the array’s length attribute. The for/each statement implicitly examines the length field to iterate the correct number of times.

The parameter passing mechanism for array variables works just like it does for other reference variables. Suppose the sumUp() method is called passing an integer array named list:

```java
int[] list;
/* Allocate and initialize list (details omitted) */
// Display the sum of all the elements in list
System.out.println(sumUp(list));
```

While in the sumUp() method, a is an alias of list. This means that modifying a within sumUp() also affects list in the calling environment.

It is not necessary to use a named array as an actual parameter. The following example makes up an anonymous array and passes it to the sumUp() method:

```java
// Display the sum of all the elements the anonymous array specified here
System.out.println(sumUp(new int[] { 10, 20, 30, 40, 50 }));
```

21.2 Copying an Array

It is important to note that an array is simply an object reference. It may seem plausible to make a copy of an array as follows:

```java
int[] a, b; // Declare two arrays
a = new int[] { 10, 20, 30 }; // Create one
b = a; // Make a copy of array a?
```

Since an array is an object reference, and since b has been assigned to a, a and b refer to the same array. Array b is an alias of a, not a copy of a. Figure 21.1 illustrates this array aliasing.

The following code can be used to make a copy of a:

```java
int[] a, b; // Declare two arrays
a = new int[] { 10, 20, 30 }; // Create one
// Really make a copy of array a
b = new int[a.length]; // Allocate b
for ( int i = 0; i < a.length; i++ ) {
    b[i] = a[i];
}
```

A new array must be created, and then each element is copied over into the new array, as shown in Figure 21.2.
int[] a, b;

First, declare the array variables

a = new int[] { 10, 20, 30 };

Next, assign a to the newly created array

b = a;

Finally, a and b refer to the same array

Since copying an array is such a common activity, the java.lang.System class has a method named arraycopy() that makes it convenient to copy all or part of the elements of one array to another array. It requires five parameters:

1. The source array; that is, the array from which the elements are to be copied
2. The starting index in the source array from which the elements will be copied
3. The destination array; that is, the array into which the elements are to be copied
4. The starting index in the destination array into which the elements will be copied
5. The number of elements to be copied

Notice that this method is quite flexible. All or part of an array may be copied into another array. The simplest case is a complete copy, as in the example above:

int[] a, b; // Declare two arrays
a = new int[] { 10, 20, 30 }; // Create one
// Make a copy of array a
b = new int[a.length]; // Allocate b
System.arraycopy(a, 0, b, 0, b.length);
In the following case part of one array is copied into part of another array:

```java
int[] a = { 10, 20, 30, 40, 50, 60, 70, 80, 90 };
int[] b = { 11, 22, 33, 44, 55, 66, 77, 88, 99 };
System.arraycopy(b, 2, a, 4, 3);
```

After executing this code the contents of `a` are:

10, 20, 30, 33, 44, 55, 80, 90

The `System.arraycopy()` method is implemented in such a way to be more efficient than equivalent code written in the Java language. Table 21.1 shows the results of five runs of `ArrayCopyBenchmark` on an array of 5,000,000 elements. On average, `System.arraycopy()` is only 2% faster than the pure Java copy code.
21.2. COPYING AN ARRAY

<table>
<thead>
<tr>
<th>Run Number</th>
<th>for Loop (msec)</th>
<th>arraycopy (msec)</th>
<th>Percent Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>137</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>137</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>115</td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>137</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>137</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>132.6</strong></td>
<td><strong>130.0</strong></td>
<td><strong>2.0</strong></td>
</tr>
</tbody>
</table>

Table 21.1: Empirical data comparing a straightforward array copy to `System.arraycopy()`. The array size is 5,000,000.

```java
if ( args.length < 1 ) {
    System.out.println("Usage:");
    System.out.println("   java ArrayCopyBenchmark <size>");
    System.exit(1);
}
int size = Integer.parseInt(args[0]);
// Initialize the source array with random values
int[] src = new int[size];
java.util.Random rand = new java.util.Random();
for ( int i = 0; i < size; i++ ) {
    src[i] = rand.nextInt(size);
}
int[] dest = new int[size];
// Copy array by hand
Stopwatch timer = new Stopwatch();
for ( int i = 0; i < size; i++ ) {
    dest[i] = src[i];
}
timer.stop();
System.out.println("Elapsed time: " + timer.elapsed());
// Copy array using System.arraycopy()
timer.start();
System.arraycopy(src, 0, dest, 0, size); timer.stop();
System.out.println("Elapsed time: " + timer.elapsed());
}
```

Listing 21.1: ArrayCopyBenchmark—compares a straightforward array copy to `System.arraycopy()`
21.3 Array Return Values

Array return values. An array can be returned by a method, as can any reference type. PrimesList (21.2) returns an array of prime numbers over a given range.

```java
import java.util.Scanner;

public class PrimesList {
    public static boolean isPrime(int n) {
        boolean result = true; // Assume no factors unless we find one
        for (int trialFactor = 2; trialFactor <= Math.sqrt(n); trialFactor++) {
            if (n % trialFactor == 0) { // Is trialFactor a factor?
                result = false;
                break; // No need to continue, we found a factor
            }
        }
        return result;
    }

    // Returns a list (array) of primes in the range start...stop
    public static int[] generatePrimes(int start, int stop) {
        // First, count how many there are
        int value = start; // Smallest potential prime number
        int count = 0; // Number of primes in the list
        while (value <= stop) {
            // See if value is prime
            if (isPrime(value)) {
                count++;
            }
            value++; // Try the next potential prime number
        }
        // Next, create an array exactly the right size
        int[] result = new int[count];
        // Next, populate the array with the primes
        count = 0;
        value = start; // Smallest potential prime number
        while (value <= stop) {
            // See if value is prime
            if (isPrime(value)) {
                result[count++] = value;
            }
            value++; // Try the next potential prime number
        }
        return result;
    }

    public static void main(String[] args) {
        int maxValue;
        Scanner scan = new Scanner(System.in);
        System.out.print("Display primes up to what value? ");
        int[] primes = generatePrimes(2, scan.nextInt());
    }
}
```
for ( int p : primes ) {
    System.out.print(p + " ");
}
System.out.println(); // Move cursor down to next line

Listing 21.2: PrimesList—Uses a method to generate a list (array) of prime numbers over a given range

The generatePrimes() method first counts the number of prime numbers so that an array of the proper size can be allocated. Then it repeats the process again filling in each position of the newly allocated array with a prime number. Note that the statement

    result[count++] = value;

uses the postincrement operator; thus, the current value of count is used as the index, then count is incremented.

21.4 Command Line Arguments

The main() method for an executable class has the general structure:

    public static void main(String[] args) {
        /* Body goes here . . . */
    }

The main() method specifies a single parameter—an array of strings. The identifier args may be replaced with any valid identifier name, but all the other names (public, static, void, main, and String) must appear as presented here. To this point we have ignored this parameter of main(), but now, armed with the knowledge of arrays, we can write even more flexible Java programs.

The command line is the sequence of text a user enters in a command shell. On a Unix/Linux system, the shell (typically bash or csh) is the command shell; on a Windows system, the DOS shell is a command shell. The command to see the files and subdirectories (subfolders) in the current directory (folder) is

    ls

on a Unix/Linux system, and

    dir

on a Windows system. Both ls and dir are considered simple command lines. Additional information can be provided to produce more complicated command lines. For example,
on a Unix/Linux system gives a “long” listing providing extra information. The command line

```
  dir /w
```

on a Windows system provides a “wide” listing of files. We say that `-l` is an argument to `ls` and `/w` is an argument to `dir`.

To execute the Java program `ArrayAverage` (20.3) from the command shell, the file `ArrayAverage.java` would first be compiled to produce the class file `ArrayAverage.class`. The Java interpreter (normally named `java`) would then be invoked as

```
  java ArrayAverage
```

This is how a Java application is typically executed from the command line.

It is possible to pass additional information to Java programs using command line arguments. The command shell (OS) uses the string array parameter in the `main()` to pass command line arguments to a Java program. `CmdLineAverage` (21.3) accepts the values to average from the command line instead of having the user enter them during the program execution.

```java
public class CmdLineAverage {
    public static void main(String[] args) {
        double sum = 0.0;
        // Get command line arguments and compute their average
        for (String s : args) {
            sum += Double.parseDouble(s);
        }
        System.out.print("The average of ");
        for (int i = 0; i < args.length - 1; i++) {
            System.out.print(args[i] + ", ");
        }
        // No comma following last element
        System.out.println(args[args.length - 1] + " is "+ sum/args.length);
    }
}
```

Listing 21.3: `CmdLineAverage`—Averages values provided on the command line

Issuing the command line

```
  java CmdLineAverage 9 3.5 0.2 100 15.3
```

yields the output
The average of 9, 3.5, 0.2, 100, 15.3 is 25.6

Figure 21.3 shows the correspondence of command line arguments to the args array elements.

java CmdLineAverage 9 3.5 0.2 100 15.3

Figure 21.3: Correspondence of command line arguments to args array elements

It is important to note that command line arguments are whitespace delimited and each argument is presented to the JVM as a String object. As shown in CmdLineAverage, if a number is provided on the command line and the argument is to be processed as a number by the Java program, then the argument must be converted from its string form to its numeric form. In CmdLineAverage, the Double.parseDouble() class method is used to convert a string argument to a double.

At last, all aspects of the declaration of the main() method have been revealed:

• **public**—the main() method can be invoked by methods outside the class
• **static**—no object need be created to execute the main() method
• **void**—main() returns no value
• **main**—the name of the method invoked when the JVM executes the class
• **String[] args**—main() accepts an array of String objects as a parameter; these strings correspond to command line arguments

In IDEs developers typically run a Java program through a menu item or hotkey. In these development environments, command line arguments are often specified in a dialog box.

### 21.5 Variable Argument Lists

It is easy to write a method that adds two integers together and returns their sum. So too, three integers, four integers, etc. up to some small practical limit. Since Java permits methods to be overloaded, the task is easy, although time consuming. OverloadedSumOfInts (21.4) provides a good starting point.

```java
public class OverloadedSumOfInts {
    public static int sum(int a, int b) {
        return a + b;
    }
    public static int sum(int a, int b, int c) {
        return a + b + c;
    }
}
```

13 March 2008 Draft © 2008 Richard L. Halterman
public static int sum(int a, int b, int c, int d) {
    return a + b + c + d;
}

public static int sum(int a, int b, int c, int d, int e) {
    return a + b + c + d + e;
}

// . . . Overload as many times as you like!

Listing 21.4: OverloadedSumOfInts—A simple class to sum integers

What if the number of integers to add exceeds the number of parameters in any of our overloaded methods? We could always use functional composition, as in

    int total = OverloadedSumOfInts.sum(OverloadedSumOfInts.sum(3, 4, -1, 18, 6),
                                        OverloadedSumOfInts.sum(2, 2, 10, -3));

but this cumbersome. We could use an array and write just one method, as is ArraySumOfInts (21.5).

    public class ArraySumOfInts {
        public static int sum(int[] a) {
            int sum = 0;
            for (int i : a) {
                sum += i;
            }
            return sum;
        }
    }

Listing 21.5: ArraySumOfInts—A “better” class to sum integers using an array

The array version is extremely flexible, but it is more awkward for the client to use. Consider the code required to sum five selected integers:

    int total = ArraySumOfInts.sum(new int[] { 3, 4, -1, 18, 6 });

The explicit array creation is necessary, but it sure would be convenient to just pass the integer arguments by themselves without the array wrapper.

Before Java 5.0, the number of arguments accepted by a method was fixed at compile time. A method declared with three formal parameters accepted only three actual parameters. As we have seen, overloaded methods provide additional flexibility, but even then the number of parameters is fixed. Java 5.0 provides a feature called varargs (the name can be traced to the C programming language) that allows method writers to leave the number of parameters open. The syntax is illustrated in VarargsSumOfInts (21.6).
public class VarargsSumOfInts {
    public static int sum(int... args) {
        int sum = 0;
        for (int i : args) {
            sum += i;
        }
        return sum;
    }
}

Listing 21.6: VarargsSumOfInts—A truly better class to sum integers using varargs

public class VarargTester {
    public static void main(String[] args) {
        System.out.println(OverloadedSumOfInts.sum(3, 4, 5, 6));
        System.out.println(ArraySumOfInts.sum(new int[] { 3, 4, 5, 6 }));
        System.out.println(VarargsSumOfInts.sum(3, 4, 5, 6));
    }
}

Listing 21.7: VarargTester—A truly better class to sum integers using varargs

Note that the body of VarargsSumOfInts is functionally identical to ArraySumOfInts; only the method signatures differ. The two methods work the same way—they process an array of ints. The new parameter specification has the following form:

\textit{typename \ldots parametername}

where \textit{typename} is any standard or programmer-defined type (primitive or reference), and \textit{parametername} is a programmer chosen parameter name. The \ldots signifies that within the method body the parameter is to be treated as an array of the type specified. In our VarargsSumOfInts example, the parameter \textit{args} is treated as an \textit{int} array. The \ldots also means that the client will send any number of parameters of the specified type. The compiler will generate code in the context of the caller that creates an array to hold these arguments and sends that new array off to the method to be processed. The client’s view is now much simpler:

\begin{verbatim}
    int total = VarargsSumOfInts.sum(3, 4, -1, 18, 6);
\end{verbatim}

The call to VarargsSumOfInts.sum() now looks like a call to a normal method that accepts five parameters. Behind the scenes, however, an array is involved, and only one true parameter is passed.

The VarargsSumOfInts.sum() method accepts any number of \textit{int} arguments, but what if you need to accept any number of arguments of mixed types? The solution is to expect any number of \textit{Object}s, as in

\begin{verbatim}
    public static void process(Object... objs) {
        /* Do something with parameters. . . */
    }
\end{verbatim}

13 March 2008 Draft  © 2008 Richard L. Halterman
Primitive types passed by the caller will be autoboxed into the appropriate wrapper objects. Of course, if nonuniform types are passed, the method body will be more tricky to write, since any type could appear at any place in the parameter list. As an example, the `System.out.printf()` method can accept any number of any types of parameters following its format string. Its signature is

```java
public PrintStream printf(String format, Object... args)
```

The code within the body of `printf()` scans the control codes within the format string to properly access the array of `Object`s that follow. That is why `printf()` can fail with a runtime error if the programmer is inconsistent with placement of control codes and arrangement of trailing parameters.

As shown in `System.out.printf()`, normal parameters can be mixed with varargs. In order for the compiler to make sense of the call of such a method, all normal parameters must precede the single varargs parameter.

### 21.6 Summary

- Add summary items here.

### 21.7 Exercises