Chapter 8

Times Square 2002

Arrays

Learning Objectives

- Describe the purpose of an array
- Describe the functionality provided by Java array types
- Trace and describe an array-type inheritance hierarchy
- List and describe the uses of single- and multidimensional arrays
- Describe how Java array objects are allocated in memory
- Describe the difference between arrays of primitive types vs. arrays of reference types
- Use the java.lang.Object and java.lang.Class classes to discover information about an array
- Demonstrate your ability to create arrays using array literals
- Demonstrate your ability to create arrays using the new operator
- Demonstrate your ability to create single-dimensional arrays
- Demonstrate your ability to create multidimensional arrays
- Describe how to access individual array elements via indexing
- Demonstrate your ability to manipulate arrays with iteration statements
- Demonstrate your ability to use the main() method's String array parameter

Introduction

The purpose of this chapter is to give you a solid foundational understanding of arrays and their usage. Since arrays enjoy special status in the Java language you will find them easy to understand and use. This chapter builds upon the material presented in chapters 6 and 7. Here you will learn how to utilize arrays in simple programs and manipulate them with program control-flow statements to yield increasingly powerful programs.

As you will soon learn, arrays are powerful data structures that can be utilized to solve many programming problems. However, even though arrays are powerful, they do have limitations. Detailed knowledge of arrays will give you the ability to judge whether an array is right for your particular application.

In this chapter you will learn the meaning of the term array, how to create and manipulate single- and multidimensional arrays, and how to use arrays in your programs. Starting with single-dimensional arrays of primitive data types, you will learn how to declare array references and how to use the new operator to dynamically create array objects. To help you better understand the concepts of arrays and their use I will show you how they are represented in memory. A solid understanding of the memory concepts behind array allocation will help you to better utilize arrays in your programs. I will then show you how to manipulate single-dimensional arrays using the program control-flow statements you learned in the previous chapter. Understanding the concepts and use of single-dimensional arrays will enable you to easily understand the concepts behind multidimensional arrays.

Along the way you will learn the difference between arrays of primitive types and arrays of reference types. I will show you how to dynamically allocate array element objects and how to call methods on objects via array element references.

Although you will learn a lot about arrays in this chapter, I have omitted some material I feel is best covered later in the book. For instance, I have postponed discussion of how to pass arrays as method arguments until you learn about classes and methods in the next chapter.

What Is An Array?

An array is a contiguous memory allocation of same-sized or homogeneous data-type elements. The word contiguous means the array elements are located one after the other in memory. The term same-sized means that each array element occupies the same amount of memory space. The size of each array element is determined by the type of objects an array is declared to contain. So, for example, if an array is declared to contain integer primitive types, each element would be the size of an integer and occupy 4-bytes. If, however, an array is declared to contain double primitive types, the size of each element would be 8-bytes. The term homogeneous is often used in place of the term same-sized to refer to objects having the same data type and therefore the same size. Figure 8-1 illustrates these concepts.

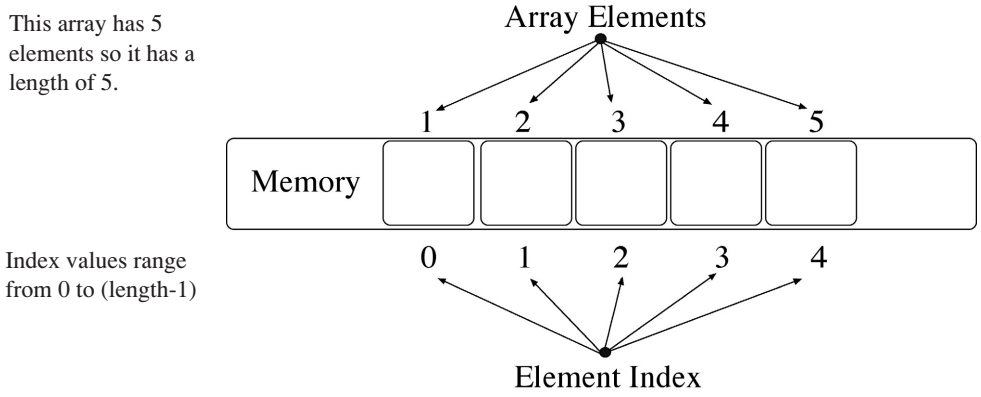


Figure 8-1: Array Elements are Contiguous and Homogeneous

Figure 8-1 shows an array of 5 elements of no particular type. The elements are numbered consecutively beginning with 1 denoting the first element and 5 denoting the last, or 5th, element in the array. Each array element is refer-

enced or accessed by its array index number. Index numbers are always one less than the element number you wish to access. For example, when you want to access the 1st element of an array you will use index number 0. To access the 2nd element of an array you will use index number 1, etc.

The number of elements an array contains is referred to as its length. The array shown in figure 8-1 contains 5 elements so it has a length of 5. The index numbers associated with this array will range from 0 to 4 (*that is 0 to (length - 1)*).

Specifying Array Types

Array elements can be primitive types, reference types, or arrays of these types. When you declare an array you must specify the type its elements will contain. Figure 8-2 illustrates this concept through the use of the array declaration and allocation syntax.

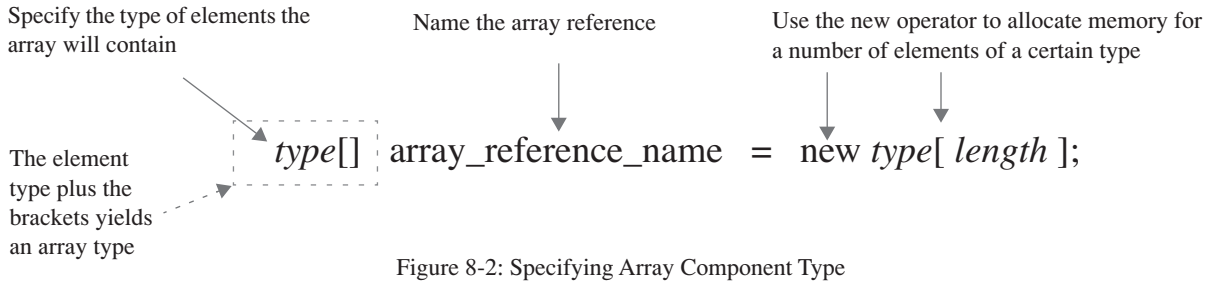


Figure 8-2: Specifying Array Component Type

Figure 8-2 shows the array declaration and allocation syntax for a single-dimensional array having a particular *type* and *length*. The declaration begins with the array element type. The elements of an array can be primitive types or reference types. Reference types can include any reference type (*class or interface*) specified in the Java API or reference types you or someone else create.

The element type is followed by a set of empty brackets. Single-dimensional arrays use one set of brackets. You will add a set of brackets for each additional dimension you want the array to have. (*I cover single- and multidimensional arrays in greater detail below.*) The element type plus the brackets yield an array type. This array type is followed by an identifier that declares the name of the array. To actually allocate memory for an array use the new operator followed by the type of elements the array can contain followed by the length of the array in brackets. The new operator returns a number representing the memory location of the newly created array object and the assignment operator assigns it to the array_reference_name.

Figure 8-2 combines the act of declaring an array and the act of creating an array object into one line of code. If required, you could declare an array in one statement and create the array in another. For example, the following line of code declares and allocates memory for a single-dimensional array of integers having length 5:

```
int[] int_array = new int[5];
```

The following line of code would simply declare an array of floats:

```
float[] float_array;
```

And this code would then allocate enough memory to hold 10 float values:

```
float_array = new float[10];
```

The following line of code would declare a two-dimensional array of boolean-type elements and allocate some memory:

```
boolean[][] boolean_array_2d = new boolean[10][10];
```

The following line of code would create a single-dimensional array of Strings:


```
String[] string_array = new String[8];
```

You will soon learn the details about single- and multidimensional arrays. If the preceding concepts seem confusing now just hang in there. By the time you complete this chapter you will be using arrays like a pro!

Quick Review

Arrays are contiguously allocated memory elements of homogeneous data types. Contiguous means the elements are arranged in memory one after the other. Homogeneous means each element of the array is of the same data type. An array containing n elements is said to have a length equal to n. Array elements are accessed via their index value which ranges from 0 to length - 1. The index value of a particular array element is always one less than the element number you wish to access. (*i.e., 1st element has index 0, 2nd element has index 1, ... , the nth element has index n-1*)

Functionality Provided By Java Array Types

The Java language has two data-type categories: primitive and reference. Array types are a special case of reference types. This means that when you create an array in Java it is an object just like a reference-type object. However, Java arrays possess special features over and above ordinary reference types. These special features make it easy to think of arrays as belonging to a distinct type category all by themselves. This section explains why Java arrays are so special.

Array-Type Inheritance Hierarchy

When you declare an array in Java you specify an array type as shown in figure 8-2 above. Just like reference types, array types automatically inherit the functionality of the java.lang.Object class. Each array type also implements the Cloneable and Serializable interfaces. Figure 8-3 gives a UML diagram of an array-type inheritance hierarchy.

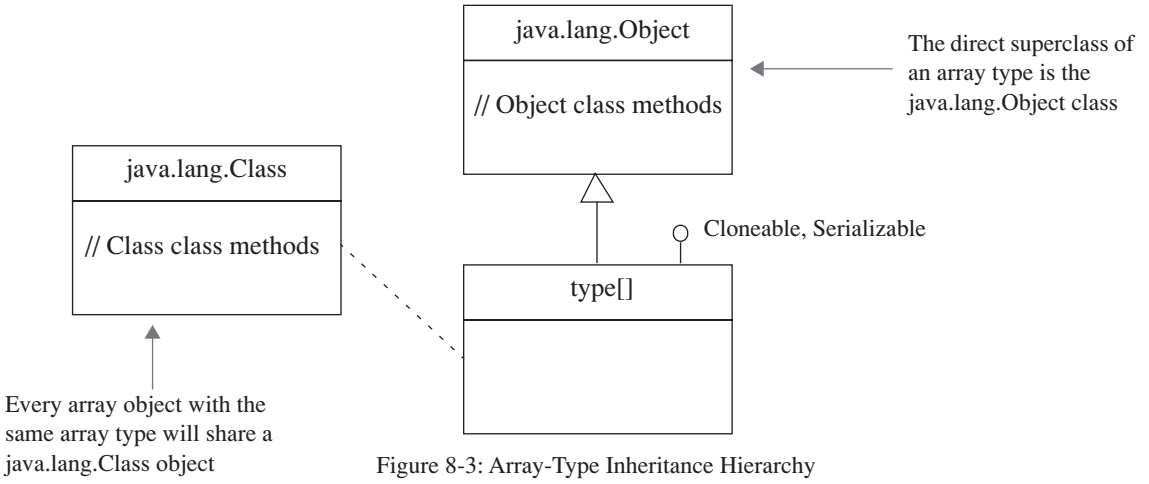


Figure 8-3: Array-Type Inheritance Hierarchy

When an array object is created in memory it will also have a corresponding java.lang.Class object. One Class object is shared by all array objects having the same array type. You will see how the Class and Object methods can be used in conjunction with array objects later in the chapter.

The java.lang.reflect.Array Class

The java.lang.reflect.Array class has of a number of static methods that can be used to manipulate array element values. I will not cover the use of the Array class in this chapter although I encourage you to explore its functionality on your own by studying its methods and what they do.

Special Properties Of Java Arrays

The following table summarizes the special properties of Java arrays:

Property	Description
Their length cannot be changed once created.	Array objects have an associated length when they are created. The length of an array cannot be changed after the array is created.

Their length can be determined via the `length` attribute	Array objects have a field named `length` that contains the value of the length of the array. To access the length attribute use the dot operator and the name of the array. For example:
	`int[] int_array = new int[5];`
	This code declares and initializes an array of integer elements with length 5. The next line of code prints the length of the int_array to the console:
	`System.out.println(int_array.length);`

Array bounds are checked by the Java virtual machine at runtime.	Any attempt to access elements of an array beyond its declared length will result in a runtime exception. This prevents mysterious data corruption bugs that can manifest themselves when misusing arrays in other languages like C or C++.
Array types directly subclass the java.lang.Object class.	Because arrays subclass Object they have the functionality of an Object.
Array types have a corresponding java.lang.Class object.	This means that you can call the java.lang.Class methods on an array object.
Elements are initialized to default values.	Primitive type array elements are initialized to the default value of the particular primitive type each element is declared to contain. Each element of an array of references is initialized to null.

Table 8-1: Java Array Properties

Quick Review

Java array types have special functionality because of their special inheritance hierarchy. Java array types directly subclass the java.lang.Object class and implement the Cloneable and Serializable interfaces. There is also one corresponding java.lang.Class object created in memory for each array type in the program.

Creating And Using Single-Dimensional Arrays

This section shows you how to declare, create, and use single-dimensional arrays of both primitive and reference types. Once you know how a single-dimensional array works you can easily apply the concepts to multidimensional arrays.

Arrays Of Primitive Types

The elements of a primitive type array can be any of the Java primitive types. These include *boolean*, *byte*, *char*, *short*, *int*, *long*, *float*, and *double*. Example 8.1 shows an array of integers being declared, created, and utilized in a short program. Figure 8-4 shows the results of running this program.

8.1 IntArrayTest.java

```
1        public class IntArrayTest {
2          public static void main(String[] args){
3             int[] int_array = new int[10];
4             for(int i=0; i<int_array.length; i++){
5                System.out.print(int_array[i] + " ");
6             }
```



```
7              System.out.println();
8           }
9        }
```

```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/IntArrayTest] swodog% java IntArrayTest
0 0 0 0 0 0 0 0 0 0
[Rick-Millers-Computer:~/desktop/IntArrayTest] swodog%
```

Figure 8-4: Results of Running Example 8.1

Example 8.1 demonstrates several important concepts. First, an array of integer type having length 10 is declared and created on line 3. The name of the array is int_array. To demonstrate that each element of the array was automatically initialized to zero, the for statement on line 4 iterates over each element of the array beginning with the first element [0] and proceeding to the last element [9], and prints each element value to the console. As you can see from looking at figure 8-4 this results in all zeros being printed to the console.

Notice how each element of int_array is accessed via an index value that appears between square brackets appended to the name of the array. (*i.e., int_array[i]*) In this example the value of i is controlled by the for loop.

How Primitive Type Array Objects Are Arranged In Memory

Figure 8-5 shows how the integer primitive type array int_array declared and created in example 8.1 is represented in memory.

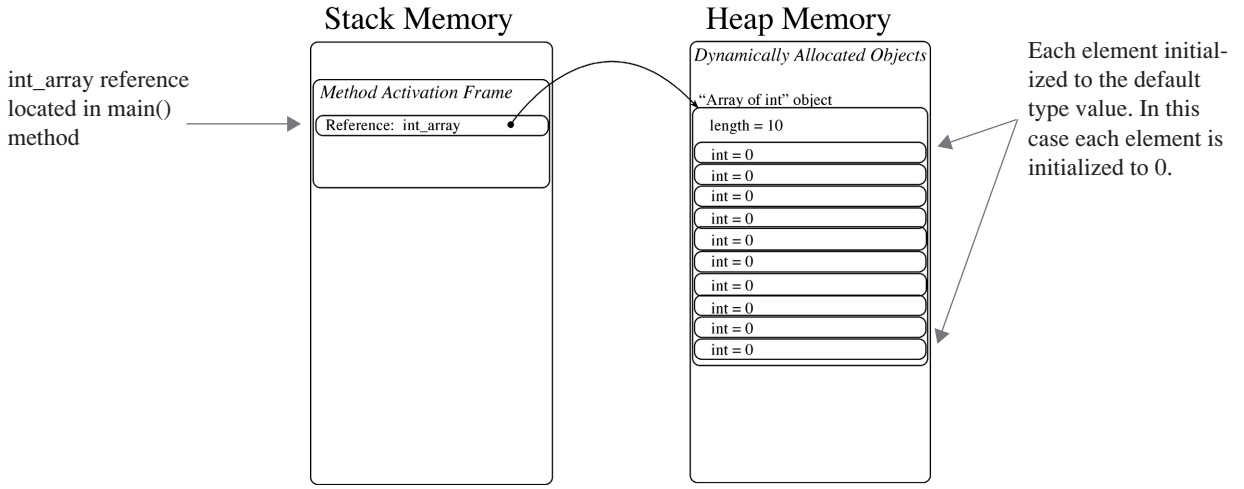


Figure 8-5: Memory Representation of Primitive Type Array int_array Showing Default Initialization

The name of the array, int_array, is a reference to an object in memory that has type "Array of int". The array object is dynamically allocated in the heap with the new operator and its memory location is assigned to the int_array reference. At the time of array object creation each element is initialized to the default value for integers which is 0. The array object's length attribute is initialized with the value of the length of the array which in this case is 10.

Let's make a few changes to example 8.1 and assign some values to the int_array elements. Example 8.2 adds another for loop that initializes each element of int_array to the value of the for loop index variable i. Figure 8-6 shows the results of running this program.

8.2 IntArrayTest.java (mod 1)

```
1        public class IntArrayTest {
2          public static void main(String[] args){
3             int[] int_array = new int[10];
4             for(int i=0; i<int_array.length; i++){
5                System.out.print(int_array[i] + " ");
6             }
7             System.out.println();
8             for(int i=0; i<int_array.length; i++){
9                int_array[i] = i;
10               System.out.print(int_array[i] + " ");
```



```
11            }
12            System.out.println();
13         }
14      }
```

```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/IntArrayTest] swodog% java IntArrayTest
0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9
[Rick-Millers-Computer:~/desktop/IntArrayTest] swodog%
```

Figure 8-6: Results of Running Example 8.2

Notice on line 9 of example 8.2 how the value of the second for loop's index variable i is assigned directly to each array element. When the array elements are printed to the console you can see that each element's value has changed except for the first which is still zero.

Figure 8-7 shows the memory representation of int_array with its new element values.

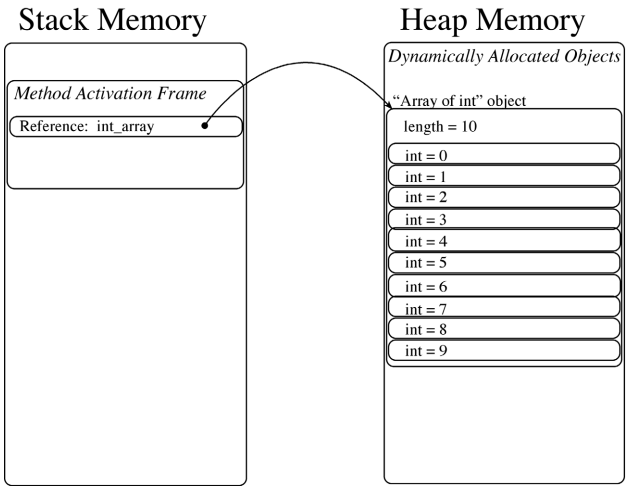


Figure 8-7: Element Values of int_array After Initialization Performed by Second for Loop

Calling Object and Class Methods on Array References

Study the code shown in example 8.3 paying particular attention to lines 15 through 18.

8.3 IntArrayTest.java (mod 2)

```
1       public class IntArrayTest {
2         public static void main(String[] args){
3            int[] int_array = new int[10];
4            for(int i=0; i<int_array.length; i++){
5               System.out.print(int_array[i] + " ");
6            }
7            System.out.println();
8            for(int i=0; i<int_array.length; i++){
9               int_array[i] = i;
10              System.out.print(int_array[i] + " ");
11           }
12           System.out.println();
13
14           /***** Calling Object & Class Methods *****/
15           System.out.println(int_array.getClass().getName());
16           System.out.println(int_array.getClass().getSuperclass());
17           System.out.println(int_array.getClass().isArray());
18           System.out.println(int_array.getClass().isPrimitive());
19        }
20      }
```

Lines 15 through 18 of example 8.3 above show how methods belonging to Object and Class can be used to get information about an array. The getClass() method belongs to java.lang.Object. It is used on each line to get the Class

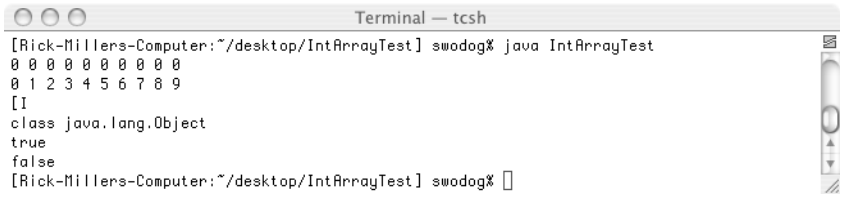


Figure 8-8: Results of Running Example 8.3

object associated with the array-type int[]. The getName(), getSuperClass(), isArray(), and isPrimitive() methods all belong to java.lang.Class.

The getName() method used on line 15 returns a String representing the name of the class. When this method is called it results in the characters "`[ I`" being printed to the console. This represents the class name of an array of integers.

On line 16 the getSuperClass() method is called to get the name of int_array's superclass. This results in the string "`class java.lang.Object`" being printed to the console indicating that the base class of int_array is Object. On line 17 the isArray() method is used to determine if int_array is an array. It returns the boolean value `true` as expected. The isPrimitive() method is used on line 18 to see if int_array is a primitive type. This returns a boolean `false` as expected since although int_array is an array of primitive types it is not itself a primitive type.

Creating Single-Dimensional Arrays Using Array Literal Values

Up to this point you have seen how memory for an array can be allocated using the new operator. Another way to allocate memory for an array and initialize the elements at the same time is to specify the contents of the array using array literal values. The length of the array is determined by the number of literal values appearing in the declaration. Example 8.4 shows two arrays being declared and created using literal values. Figure 8-9 shows the results of running this program.

8.4 ArrayLiterals.java

```
1         public class ArrayLiterals {
2           public static void main(String[] args){
3             int[] int_array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
4             double[] double_array = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0};
5
6             for(int i = 0; i < int_array.length; i++){
7                System.out.print(int_array[i] + " ");
8             }
9             System.out.println();
10            System.out.println(int_array.getClass().getName());
11            System.out.println(int_array.getClass().isArray());
12
13            System.out.println();
14
15            for(int i = 0; i < double_array.length; i++){
16               System.out.print(double_array[i] + " ");
17            }
18            System.out.println();
19            System.out.println(double_array.getClass().getName());
20            System.out.println(double_array.getClass().isArray());
21          }
22        }
```

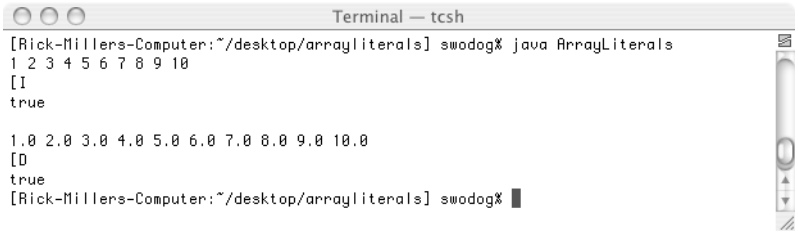


Figure 8-9: Results of Running Example 8.4

Example 8.4 declares and initializes two arrays using array literal values. On line 3 an array of integer primitive types named int_array is declared. The elements of the array are initialized to the values that appear between the braces. Each element's literal value is separated by a comma. The length of the array is determined by the number of literal values appearing between the braces. The length of int_array is 10.

On line 4 an array of double primitive types named double_array is declared and initialized with double literal values. The contents of both arrays are printed to the console. Object and Class methods are then used to determine the characteristics of each array and the results are printed to the console.

Differences Between Arrays Of Primitive Types And Arrays Of Reference Types

The key difference between arrays of primitive types and arrays of reference types is that primitive type values can be directly assigned to primitive type array elements. The same is not true for reference type elements. In an array of reference types, each element is a reference to an object in memory. When you create an array of references in memory you are not automatically creating each element's object. Each reference element is automatically initialized to null and the object you want it to point to must be explicitly created. (*Or the object must already exist somewhere in memory and be reachable*.) To illustrate these concepts I will use an array of Objects. Example 8.5 gives the code for a short program that creates and uses an array of Objects. Figure 8-10 shows the results of running this program.

8.5 ObjectArray.java

```
1        public class ObjectArray {
2         public static void main(String[] args){
3            Object[] object_array = new Object[10];
4
5            object_array[0] = new Object();
6            System.out.println(object_array[0].getClass());
7            System.out.println(object_array[0].toString());
8            System.out.println();
9
10           object_array[1] = new Object();
11           System.out.println(object_array[1].getClass());
12           System.out.println(object_array[1].toString());
13           System.out.println();
14
15           for(int i = 2; i < object_array.length; i++){
16               object_array[i] = new Object();
17               System.out.println(object_array[i].getClass());
18               System.out.println(object_array[i].toString());
19               System.out.println();
20           }
21         }
22       }
```

Referring to example 8.5, on line 3 an array of Objects of length 10 is declared and created. After line 3 executes the object_array reference will point to an array of Objects in memory with each element initialized to null as is shown in figure 8-11 below.

On line 5 a new Object object is created and its memory location is assigned to the Object reference located in object_array[0]. The memory picture now looks like figure 8-12 below. Lines 6 and 7 call the getClass() and the toString() methods on the object pointed to by object_array[0].

The execution of line 10 results in the creation of another Object object in memory. The memory picture now looks like figure 8.13. The for statement on line 15 creates the remaining Object objects and assigns their memory locations to the remaining object_array reference elements. Figure 8.14 shows the memory picture after the for statement completes execution.

Now that you know the difference between primitive and reference type arrays let's see some single-dimensional arrays being put to good use.

Single-dimensional Arrays In Action

This section offers several example programs showing how single-dimensional arrays can be used in programs. These programs represent an extremely small sampling of the usefulness arrays afford.

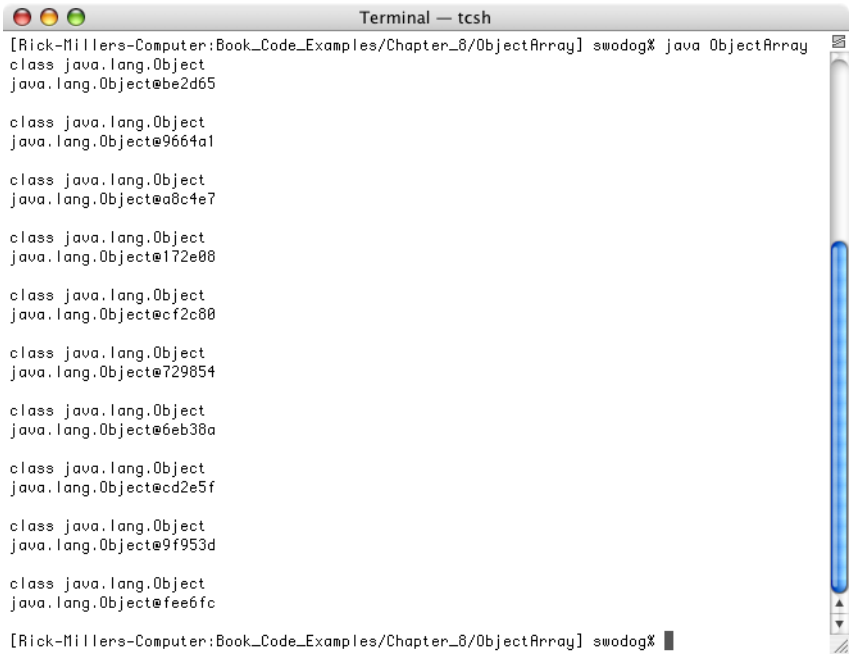


Figure 8-10: Results of Running Example 8.5

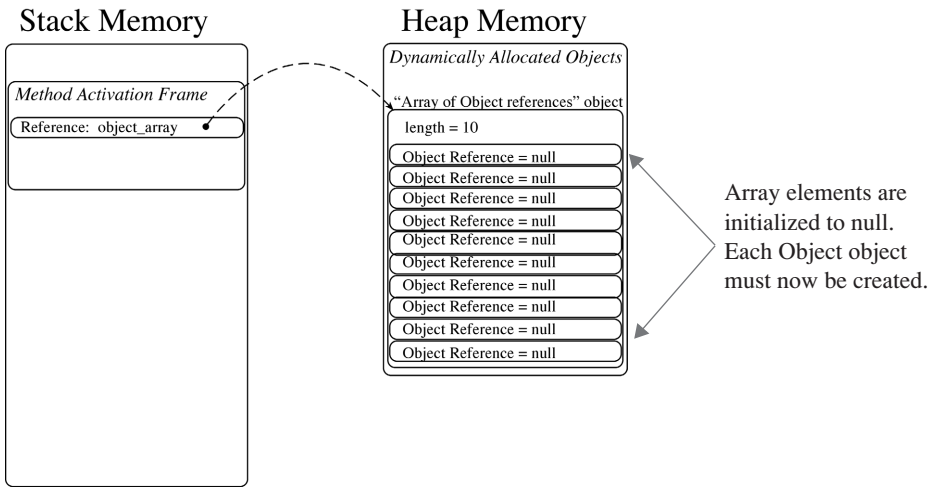


Figure 8-11: State of Affairs After Line 3 of Example 8.5 Executes

Message Array

One handy use for an array is to store a collection of String messages for later use in a program. Example 8.6 shows how such an array might be utilized. Figure 8-15 shows the results of running this program twice.

8.6 MessageArray.java

```
1         import java.io.*;
2
3         public class MessageArray {
4           public static void main(String[] args){
5              BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
6              String name = null;
7              int int_val = 0;
8
9              String[] messages = {"Welcome to the Message Array Program",
10                                  "Please enter your name: ",
11                                  ", please enter an integer: ",
```

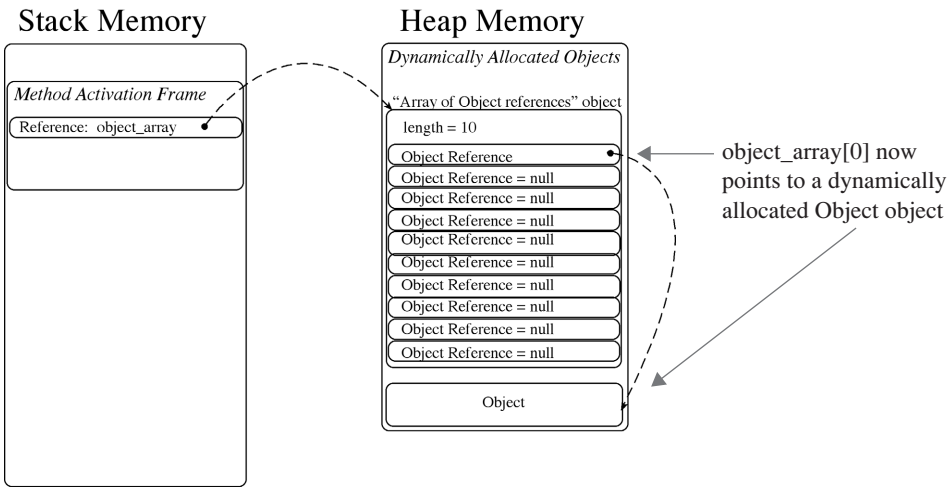



Figure 8-12: State of Affairs After Line 5 of Example 8.5 Executes.

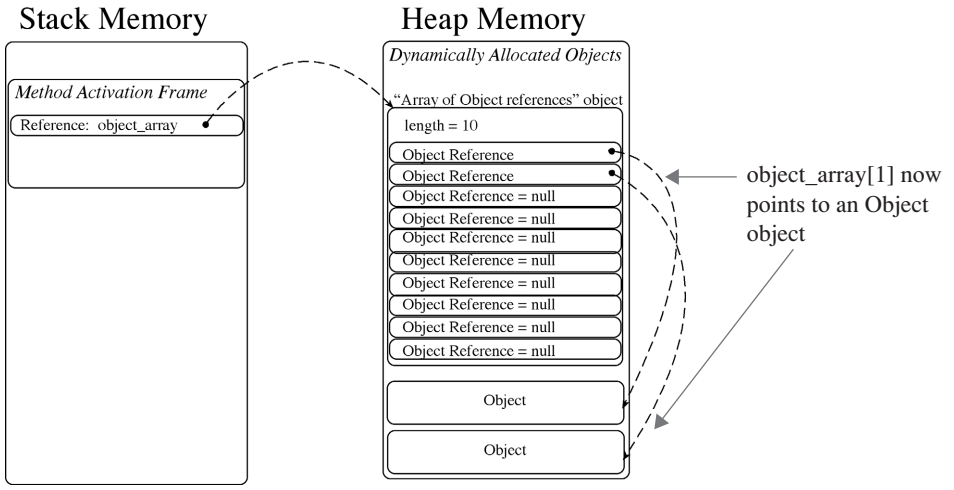


Figure 8-13: State of Affairs After Line 10 of Example 8.5 Executes

```
12                              "You did not enter an integer!",
13                              "Thank you for running the Message Array program",
14                              "Console read error!"};
15
16            final int WELCOME_MESSAGE     = 0;
17            final int ENTER_NAME_MESSAGE  = 1;
18            final int ENTER_INT_MESSAGE   = 2;
19            final int INT_ERROR           = 3;
20            final int THANK_YOU_MESSAGE   = 4;
21            final int CONSOLE_READ_ERROR  = 5;
22
23          System.out.println(messages[ WELCOME_MESSAGE] );
24          System.out.print(messages[ ENTER_NAME_MESSAGE] );
25          try{
26              name = console.readLine();
27             }catch(Exception e){  System.out.println(messages[ CONSOLE_READ_ERROR] ); }
28
29          System.out.print(name + messages[ ENTER_INT_MESSAGE] );
30
31          try{
32              int_val = Integer.parseInt(console.readLine());
33             }catch(NumberFormatException nfe) {  System.out.println(messages[ INT_ERROR] ); }
34              catch(IOException ioe) {  System.out.println(messages[ CONSOLE_READ_ERROR] ); }
35
36          System.out.println(messages[ THANK_YOU_MESSAGE] );
37        }
38      }
```

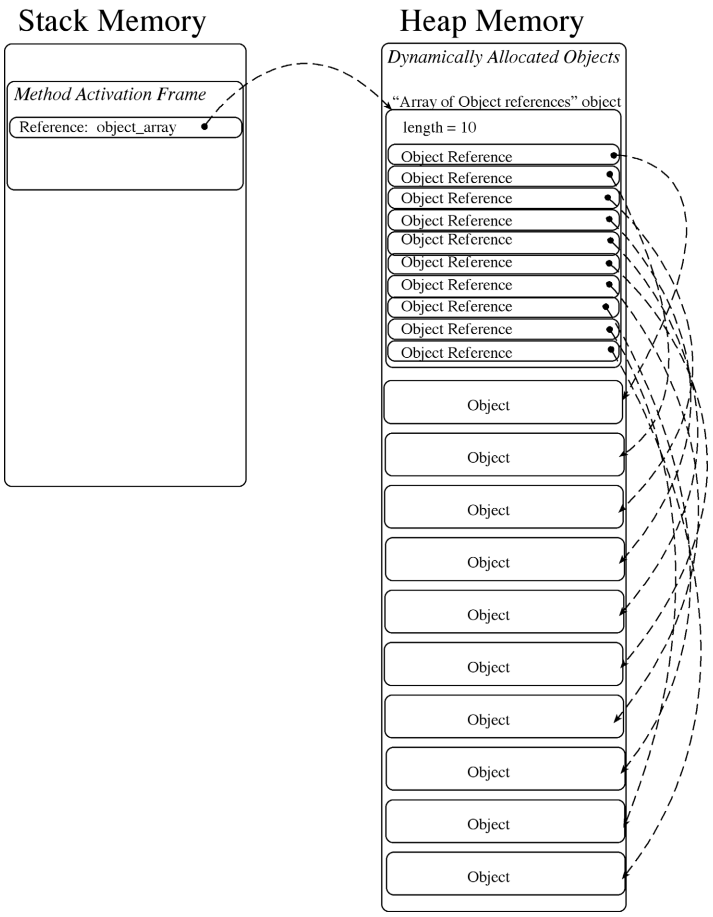



Figure 8-14: Final State of Affairs: All object_array Elements Point to an Object object

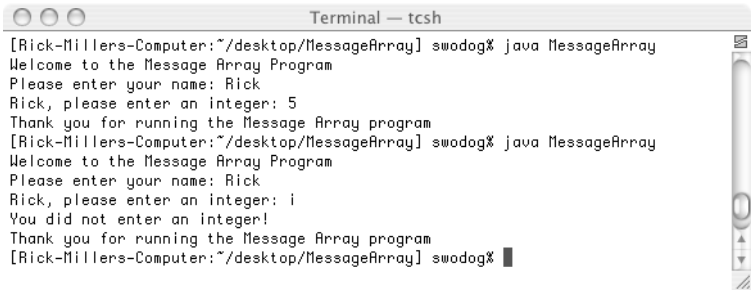


Figure 8-15: Results of Running Example 8.6

Example 8.6 creates a single-dimensional array of Strings named messages. It initializes each String element using String literals. (*Strings are given special treatment by the Java language and can be initialized using String literal values*.) On lines 16 through 21 an assortment of constants are declared and initialized. These constants are used to index the messages array as is shown on lines 23 and 24.

The program simply asks the user to enter their name followed by a request for them to enter an integer value. Since the readLine() method may throw an IOException it must be enclosed in a try/catch block. Next, the user is prompted to enter an integer value. Now two things may go wrong. Either, the readLine() method may throw an IOException or the Integer.parseInt() method may throw a NumberFormatException. Notice the use of two catch blocks to check for each type of exception. Also notice in figure 8-15 the effects of entering a bad integer value.

Calculating Averages

The program given in example 8.7 will calculate class grade averages.

8.7 Average.java

```
1        import java.io.*;
2        import java.text.*;
3
4        public class Average {
5          public static void main(String[] args){
6            BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
7            double[] grades      = null;
8            double   total       = 0;
9            double   average     = 0;
10           int      grade_count = 0;
11           NumberFormat nf = NumberFormat.getInstance();
12
13           nf.setMaximumFractionDigits(2);
14           System.out.println("Welcome to Grade Averager");
15           System.out.print("Please enter the number of grades to enter: ");
16           try{
17               grade_count = Integer.parseInt(console.readLine());
18           } catch(NumberFormatException nfe) { System.out.println("You did not enter a number!"); }
19             catch(IOException ioe) { System.out.println("Problem reading console!"); }
20
21           if(grade_count > 0){
22               grades = new double[grade_count];
23               for(int i = 0; i < grade_count; i++){
24                   System.out.print("Enter grade " + (i+1) + ": ");
25                   try{
26                       grades[i] = Double.parseDouble(console.readLine());
27                   } catch(NumberFormatException nfe) { System.out.println("You did not enter a number!"); }
28                     catch(IOException ioe) { System.out.println("Problem reading console!"); }
29               } //end for
30
31
32               for(int i = 0; i < grade_count; i++){
33                   total += grades[i];
34               } //end for
35
36               average = total/grade_count;
37               System.out.println("Number of grades entered: " + grade_count);
38               System.out.println("Grade average:            " + nf.format(average));
39           }//end if
40         } //end main
41       }// end Average class definition
```

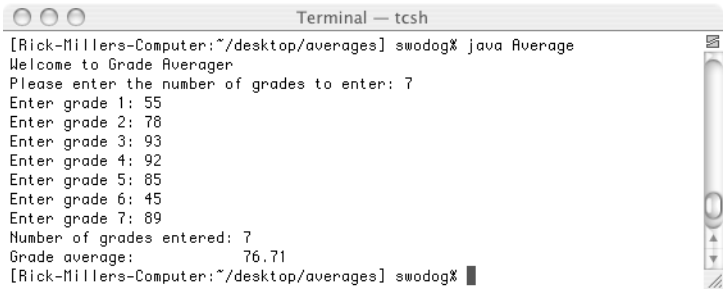


Figure 8-16: Results of Running Example 8.7

Referring to example 8.7 — an array reference of double primitive types named grades is declared on line 7 and initialized to null. On lines 8 through 10 several other program variables are declared and initialized. Take special note of what's happening on line 11. Here a java.text.NumberFormat reference named nf is declared and initialized using the static method getInstance() provided by the NumberFormat class. Essentially, the NumberFormat class provides you with the capability to format numeric output in various ways. The NumberFormat reference nf is then used on line 13 to set the number of decimal places to display when writing floating point values. I have set the number of decimal places to 2 by calling the setMaximumFractionDigits() method with an argument of 2.

The program then prompts the user to enter the number of grades. If this number is greater than 0 then it is used on line 22 to create the grades array. The program then enters a for loop on line 23, reads each grade from the console, converts it to a double, and assigns it to the i^{th} element of the grades array.

After all the grades are entered into the array the grades are summed in the for loop on line 32 and the average is calculated on line 36. Notice how the NumberFormat reference nf is used on line 38 to properly format the double value contained in the average variable.

Histogram: Letter Frequency Counter

Letter frequency counting is an important part of deciphering messages that were encrypted using monalphabetic substitution. Example 8.8 gives the code for a program that counts the occurrences of each letter appearing in a text string and prints a letter frequency display to the console. The program ignores all characters except the 26 letters of the alphabet. Figure 8-17 gives the results of running this program with a sample line of text.

8.8 Histogram.java

```
1         import java.io.*;
2
3         public class Histogram {
4           public static void main(String[] args){
5             BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
6             int letter_frequencies[] = new int[26];
7             final int A = 0, B = 1, C = 2, D = 3, E = 4, F = 5, G = 6,
8                       H = 7, I = 8, J = 9, K = 10, L = 11, M = 12, N = 13,
9                       O = 14, P = 15, Q = 16, R = 17, S = 18, T = 19, U = 20,
10                      V = 21, W = 22, X = 23, Y = 24, Z = 25;
11            String input_string = null;
12
13            System.out.print("Enter a line of characters: ");
14            try {
15                input_string = console.readLine().toUpperCase();
16                } catch(IOException ioe) { System.out.println("Problem reading console!"); }
17
18            if(input_string != null){
19              for(int i = 0; i < input_string.length(); i++){
20                switch(input_string.charAt(i)){
21                 case 'A': letter_frequencies[A]++;
22                           break;
23                 case 'B': letter_frequencies[B]++;
24                           break;
25                case 'C': letter_frequencies[C]++;
26                           break;
27                case 'D': letter_frequencies[D]++;
28                           break;
29                case 'E': letter_frequencies[E]++;
30                           break;
31                case 'F': letter_frequencies[F]++;
32                           break;
33                case 'G': letter_frequencies[G]++;
34                           break;
35                case 'H': letter_frequencies[H]++;
36                           break;
37                case 'I': letter_frequencies[I]++;
38                           break;
39                case 'J': letter_frequencies[J]++;
40                           break;
41                case 'K': letter_frequencies[K]++;
42                           break;
43                case 'L': letter_frequencies[L]++;
44                           break;
45                case 'M': letter_frequencies[M]++;
46                           break;
47                case 'N': letter_frequencies[N]++;
48                           break;
49                case 'O': letter_frequencies[O]++;
50                           break;
51                case 'P': letter_frequencies[P]++;
52                           break;
53                case 'Q': letter_frequencies[Q]++;
54                           break;
55                case 'R': letter_frequencies[R]++;
56                           break;
57                case 'S': letter_frequencies[S]++;
58                           break;
59                case 'T': letter_frequencies[T]++;
60                           break;
61                case 'U': letter_frequencies[U]++;
62                           break;
63                case 'V': letter_frequencies[V]++;
64                           break;
```

```
65                case 'W': letter_frequencies[ W] ++;
66                          break;
67                case 'X': letter_frequencies[ X] ++;
68                          break;
69                case 'Y': letter_frequencies[ Y] ++;
70                          break;
71                case 'Z': letter_frequencies[ Z] ++;
72                          break;
73                default :
74                } //end switch
75              } //end for
76
77            for(int i = 0; i < letter_frequencies.length; i++){
78               System.out.print((char)(i + 65) + ": ");
79               for(int j = 0; j < letter_frequencies[ i] ; j++){
80                System.out.print('*');
81               } //end for
82               System.out.println();
83            } //end for
84
85          } //end if
86        } // end main
87      } // end Histogram class definition
```

```
Terminal — tcsh
[Rick-Millers-Computer:~/desktop/histogram] swodog% java Histogram
Enter a line of characters: Oooooooh if you loved Java like I love Java then we'd both love Java too...
A: ******
B: *
C:
D: **
E: ******
F: *
G:
H: ***
I: ***
J: ***
K: *
L: ****
M:
N: *
O: ************
P:
Q:
R:
S:
T: ***
U: *
V: ******
W: *
X:
Y: *
Z:
[Rick-Millers-Computer:~/desktop/histogram] swodog%
```

Figure 8-17: Results of Running Example 8.8

Referring to example 8.8 — on line 6 an integer array named letter_frequencies is declared and initialized to contain 26 elements, one for each letter of the alphabet. On lines 7 through 10 several constants are declared and initialized. The constants, named A through Z, are used to index the letter_frequencies array later in the program. On line 11 a String reference named input_string is declared and initialized to null.

The program then prompts the user to enter a line of characters. The program reads this line of text and converts it all to upper case using the toUpperCase() method of the String class. Most of the work is done within the body of the if statement that starts on line 18. If the input_string is not null then the for loop will repeatedly execute the switch statement, testing each letter of input_string and incrementing the appropriate letter_frequencies element.

Take special note on line 19 of how the length of the input_string is determined using the String class's length() method. A String object is not an array but the methods length() and charAt() allow a String object to be manipulated on a read-only basis in similar fashion.

Quick Review

Single-dimensional arrays have one dimension — length. You can get an array's length by accessing its length attribute. Arrays can have elements of primitive or reference types. An array type is created by specifying the type

name of array elements followed by one set of brackets [ ]. Use the java.lang.Object and java.lang.Class methods to get information about an array.

Each element of an array is accessed via an index value contained in a set of brackets. Primitive-type element values can be directly assigned to array elements. When an array of primitive types is created each element is initialized to a default value that's appropriate for the element type. Each element of an array of references is initialized to null. Each object each reference element points to must be dynamically created at some point in the program.

Creating And Using Multidimensional Arrays

Java multidimensional arrays are arrays of arrays. For instance, a two-dimensional array is an array of arrays. A three-dimensional array is an array of arrays of arrays, etc. In general, when the elements of an array are themselves arrays, the array is said to be multidimensional. Everything you already know about single-dimensional arrays applies to multidimensional arrays. This means you are already over half way to complete mastery of multidimensional arrays. All you need to know now is how to declare them and use them in your programs.

Multidimensional Array Declaration Syntax

You can declare arrays of just about any dimension. However, the most common multidimensional arrays you will see in use are of two or three dimensions. The syntax for declaring and creating a two-dimensional array is shown in figure 8-18.

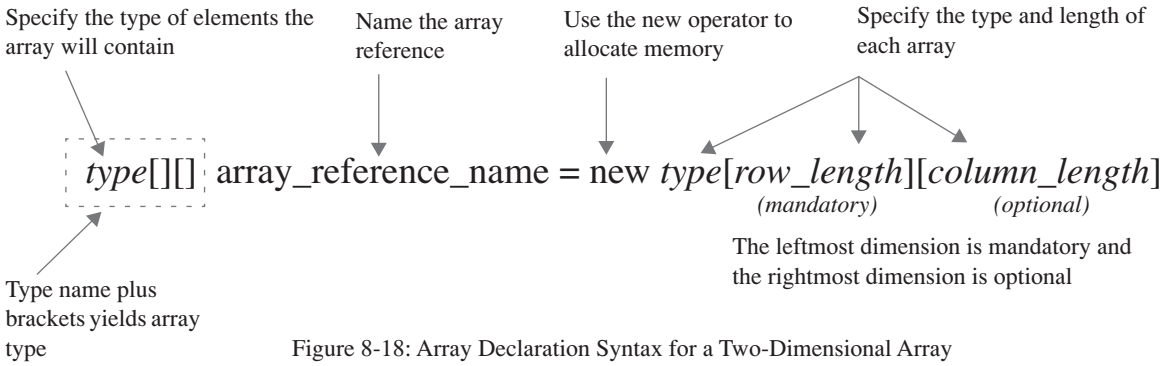


Figure 8-18: Array Declaration Syntax for a Two-Dimensional Array

As figure 8-18 illustrates there are a few new things to keep in mind when declaring multidimensional arrays. First, you will include an extra set of brackets for each additional dimension you want to give your array. Since this is an example of a two-dimensional array there is one extra set of brackets appended to the array type.

When you use the new operator to allocate memory for the array, only the leftmost dimension(s) are mandatory while the rightmost dimension is optional. This enables you to create jagged arrays. (*i.e., arrays of different lengths*)

Let's take a look at an example of a multidimensional array declaration. The following line of code declares and creates a two-dimensional array of integers with the dimensions 10 by 10:

```
int[][] int_2d_array = new int[10][10];
```

You could think of this array as being arranged by rows and columns as is shown in figure 8-19.

Each row is a reference to an integer array and there are 10 rows (int[**10**][]) and each column is an element of a 10 integer array (int[10][**10**]). Each element of int_2d_array can be accessed using a double set of brackets that indicate the row and column of the desired element. For example, the 3rd element of the 3rd array can be found at int_2d_array[2][2], the 1st element of the 6th array can be found at int_2d_array[5][0], the 6th element of the 6th array can be found at int_2d_array[5][5], and the 9th element of the 10th array can be found at int_2d_array[9][8].

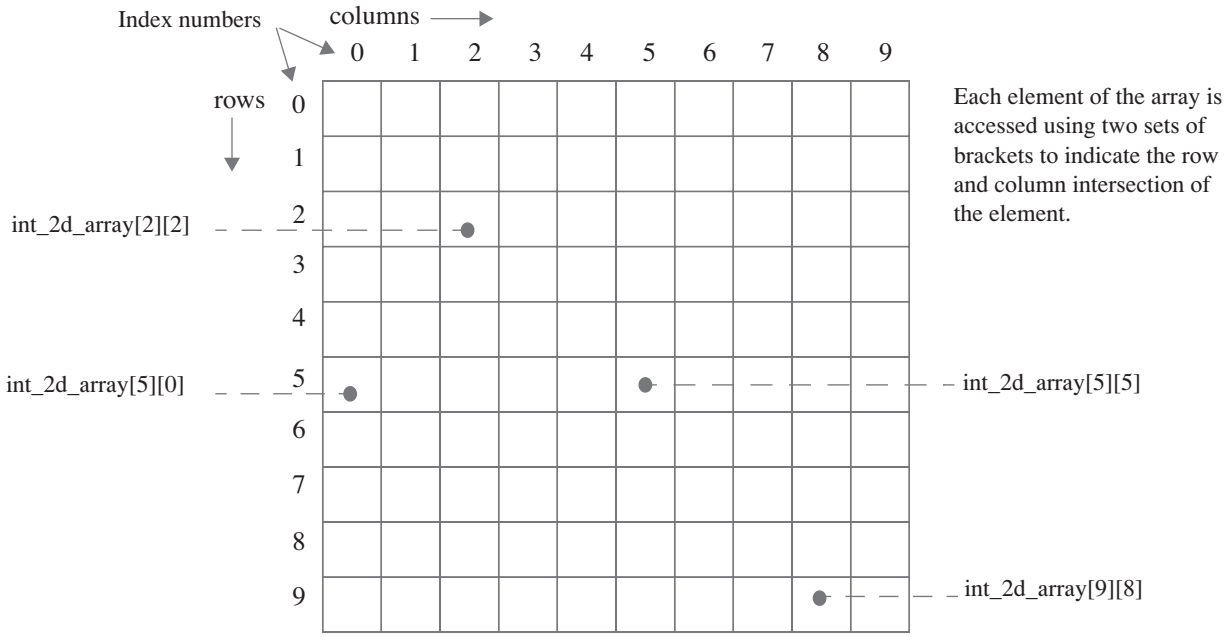


Figure 8-19: A Two Dimensional Array with Dimensions 10 by 10

Example 8.9 is a short program that will let you experiment with creating two dimensional arrays of different dimensions. Figure 8-20 shows this program in action using arrays of various dimensions.

8.9 MultiArrays.java

```
1         public class MultiArrays {
2           public static void main(String[ ] args){
3              int rows = Integer.parseInt(args[ 0] );
4              int cols = Integer.parseInt(args[ 1] );
5
6              int[ ][ ] int_2d_array = new int[ rows][ cols] ;
7
8              System.out.println(int_2d_array.getClass());
9
10             for(int i = 0; i<int_2d_array.length; i++){
11               for(int  j = 0; j<int_2d_array[ i] .length; j++){
12                System.out.print(int_2d_array[ i][ j] );
13               }
14                System.out.println();
15             }
16          }
17        }
```

Figure 8-20: Results of Running Example 8.9

Referring to example 8.9 — the getClass() method is used on line 8 to determine int_2d_array's class type. As you can see from figure 8-20 its class type is "`[ [ I`" meaning array of int arrays. When this program is run the number of rows desired is entered on the command line followed by the number of columns desired. It simply creates an integer array with the desired dimensions and prints its contents to the console in row order.

Memory Representation Of A Two Dimensional Array

Figure 8-21 offers a memory representation of the int_2d_array used in example 8.9 with the dimensions of 2 rows and 10 columns. As you can see, the int_2d_array reference declared in the main() method points to an array of integer arrays. This array of integer arrays represents the rows. Each "row" element is a reference to an array of integers. The length of each of these integer arrays represents the number of columns contained in the two-dimensional array.

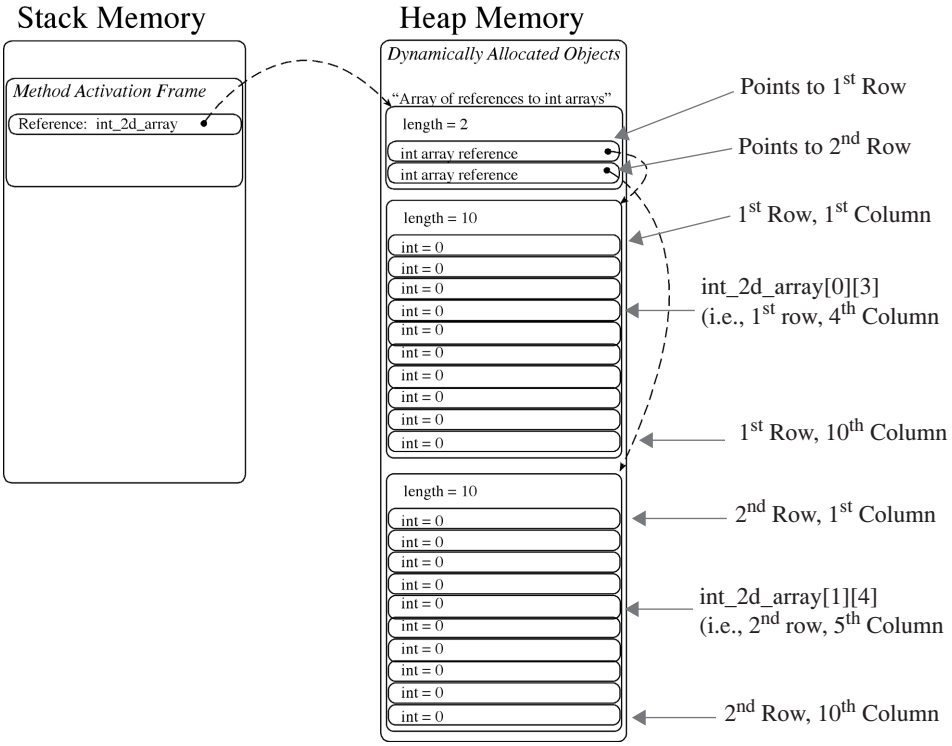


Figure 8-21: Memory Representation of int_2d_array with 2 Rows and 10 Columns

Creating Multidimensional Arrays Using Array Literals

Multidimensional arrays can be created using array literals. Example 8.10 is a short program that creates several two-dimensional arrays using array literal values. Figure 8-22 shows the results of running this program.

8.10 TwoDArrayLiterals.java

```
1         public class TwoDArrayLiterals {
2           public static void main(String[] args){
3             /************** ragged int array *************************/
4             int[][] int_2d_ragged_array = {{1,2},
5                                            {1,2,3,4},
6                                            {1,2,3},
7                                            {1,2,3,4,5,6,7,8}};
8
9
10            for(int i = 0; i < int_2d_ragged_array.length; i++){
11              for(int j = 0; j < int_2d_ragged_array[i].length; j++){
12                System.out.print(int_2d_ragged_array[i][j] + " ");
13              }
14              System.out.println();
15            }
```



```
16
17          /************* ragged String array ***********************/
18          String[][] string_2d_ragged_array = {{"Now ", "is ", "the ", "time "},
19                                               {"for ", "all ", "good men "},
20                                               {"to come to ", "the aid "},
21                                               {"of their country!"}};
22
23          for(int i = 0; i < string_2d_ragged_array.length; i++){
24            for(int j = 0; j < string_2d_ragged_array[i].length; j++){
25              System.out.print(string_2d_ragged_array[i][j]);
26            }
27            System.out.println();
28          }
29        }
30      }
```

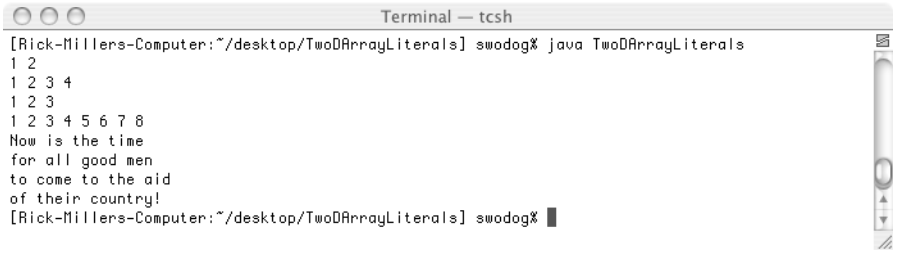


Figure 8-22: Results of Running Example 8.10

Ragged Arrays

The arrays created in example 8.10 are known as ragged arrays. A ragged array is a multidimensional array having row lengths of varying sizes. Ragged arrays are easily created using array literals. That's because each row dimension can be explicitly set when the array is declared and created at the same time.

Another way to create a ragged array is to omit the final dimension at the time the array is created, and then dynamically create each array of the final dimension. Study the code shown in example 8.11.

8.11 RaggedArray.java

```
1       import java.io.*;
2
3       public class RaggedArray {
4         public static void main(String[] args){
5          BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
6          int[][] int_2d_ragged_array = new int[5][]; //rightmost dimension omitted
7          int dimension = 0;
8
9          /*************** get dimension of each ragged array ***************/
10         for(int i = 0; i < int_2d_ragged_array.length; i++){
11            System.out.print("Enter the ragged array dimension: ");
12            dimension = 0;
13            try {
14                dimension = Integer.parseInt(console.readLine());
15             }catch(NumberFormatException nfe){ System.out.println("Bad number! Dimension being set to 3");
16                                                dimension = 3; }
17            catch(IOException ioe){ System.out.println("Problem reading console! Dimension being set to 3");
18                                    dimension = 3; }
19
20            int_2d_ragged_array[i] = new int[dimension];
21         }//end for
22
23         /******************* print contents of array *********************/
24         for(int i = 0; i < int_2d_ragged_array.length; i++){
25           for(int j = 0; j < int_2d_ragged_array[i].length; j++){
26             System.out.print(int_2d_ragged_array[i][j]);
27           } // end inner for
28           System.out.println();
29         }//end outer for
30
31       }//end main
32      }//end class
```

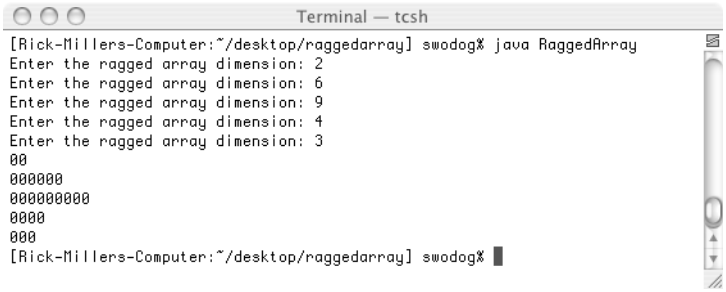


Figure 8-23: Results of Running Example 8.11

The important point to note about example 8.11 occurs on line 6. Notice how the rightmost array dimension is omitted when the array is created with the new operator. Since the final array dimension is missing, the length of each final array must be explicitly created in the program. This is done in the first for loop that begins on line 10. Each final array is actually created on line 20.

Multidimensional Arrays In Action

The example presented in this section shows how single- and multidimensional arrays can be used together to achieve a high degree of functionality.

Weighted Grade Tool

Example 8.12 gives the code for a class named WeightedGradeTool. The program calculates a student's final grade based on weighted grades. Figure 8-24 shows the results of running this program.

8.12 WeightedGradeTool.java

```
1         import java.io.*;
2
3         public class WeightedGradeTool {
4           public static void main(String[] args){
5             BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
6             double[][]   grades   = null;
7             double[]  weights  = null;
8             String[]  students = null;
9             int student_count = 0;
10            int grade_count = 0;
11            double final_grade = 0;
12
13            System.out.println("Welcome to Weighted Grade Tool");
14
15           /*************** get student count ********************/
16            System.out.print("Please enter the number of students: ");
17            try{
18               student_count = Integer.parseInt(console.readLine());
19               }catch(NumberFormatException nfe){  System.out.println("That was not an integer!");
20                                                  System.out.println("Student count will be set to 3.");
21                                                  student_count = 3; }
22                catch(IOException ioe){  System.out.println("Trouble reading from the console!");
23                                         System.out.println("Student count will be set to 3.");
24                                         student_count = 3; }
25
26           if(student_count > 0){
27              students = new String[student_count];
28              /***************** get student names **********************/
29              getNames: for(int i = 0; i < students.length; i++){
30                           System.out.print("Enter student name: ");
31                           try{
32                              students[i]  = console.readLine();
33                               } catch(IOException ioe){  System.out.println("Problem reading console!");
34                                                          System.exit(0); }
35              }//end getNames for
36
37              /*************** get number of grades per student ***************************/
38              System.out.print("Please enter the number of grades to average: ");
39              try{
40               grade_count = Integer.parseInt(console.readLine());
41               }catch(NumberFormatException nfe){  System.out.println("That was not an integer!");
```



```
42                                          System.out.println("Grade count will be set to 3.");
43                                          grade_count = 3; }
44              catch(IOException ioe){  System.out.println("Trouble reading from the console!");
45                                       System.out.println("Grade count will be set to 3.");
46                                       grade_count = 3; }
47
48           /****************** get raw grades ****************************/
49           grades = new double[ student_count][ grade_count];
50           getGrades: for(int i = 0; i < grades.length; i++){
51                        System.out.println("Enter raw grades for " + students[ i]);
52                        for(int j = 0; j < grades[ i].length; j++){
53                          System.out.print("Grade " + (j+1) + ": ");
54                          try{
55                              grades[ i][ j] = Double.parseDouble(console.readLine());
56                           }catch(NumberFormatException nfe){  System.out.println("That was not a double!");
57                                                              System.out.println("Grade will be set to 100");
58                                                              grades[ i][ j] = 100; }
59                           catch(IOException ioe){  System.out.println("Trouble reading from the console!");
60                                                    System.out.println("Grade will be set to 100.");
61                                                    grades[ i][ j] = 100; }
62
63                        }//end inner for
64                     }// end getGrades for
65
66          /***************** get grade weights ********************/
67          weights = new double[ grade_count];
68          System.out.println("Enter grade weights. Make sure they total 100%");
69          getWeights: for(int i = 0; i < weights.length; i++){
70                        System.out.print("Weight for grade " + (i + 1) + ": ");
71                        try{
72                           weights[ i] = Double.parseDouble(console.readLine());
73                           }catch(NumberFormatException nfe){  System.out.println("That was not a double!");
74                                                               System.out.println("Program will exit!");
75                                                               System.exit(0); }
76                           catch(IOException ioe){  System.out.println("Trouble reading from the console!");
77                                                    System.out.println("Program will exit!");
78                                                    System.exit(0); }
79                     }//end getWeights for
80
81           /****************** calculate weighted grades *******************/
82           calculateGrades: for(int i = 0; i < grades.length; i++){
83                              for(int j = 0; j < grades[ i].length; j++){
84                                 grades[ i][ j] *= weights[ j];
85                              }//end inner for
86                            }//end calculateGrades for
87
88           /***************** calculate and print final grade *********************/
89           finalGrades: for(int i = 0; i < grades.length; i++){
90                           System.out.println("Weighted grades for " + students[ i] + ": ");
91                           final_grade = 0;
92                            for(int j = 0; j < grades[ i].length; j++){
93                              final_grade += grades[ i][ j];
94                           System.out.print(grades[ i][ j] + " ");
95                            }//end inner for
96                           System.out.println(students[ i] +"'s final grade is: " + final_grade );
97                         }//end averageGrades for
98          }// end if
99        }// end main
100     }// end class
```

Quick Review

Multidimensional arrays are arrays having more than one dimension. Multidimensional arrays in Java are arrays of arrays. An understanding of single-dimensional Java arrays leads to quick mastery of multidimensional arrays.

To declare a multidimensional array simply add a set of brackets to the array element type for each dimension required in the array. Multidimensional arrays can contain any number of dimensions, however, arrays of two and three dimensions are most common.

Multidimensional arrays can be created using array literal values. The use of array literals enables the initialization of each array element at the point of declaration.

Ragged arrays are multidimensional arrays having final element arrays of various lengths. Ragged arrays can be easily created with array literals. They can also be created by omitting the final dimension at the time of array creation and then dynamically creating each final array with a unique length.

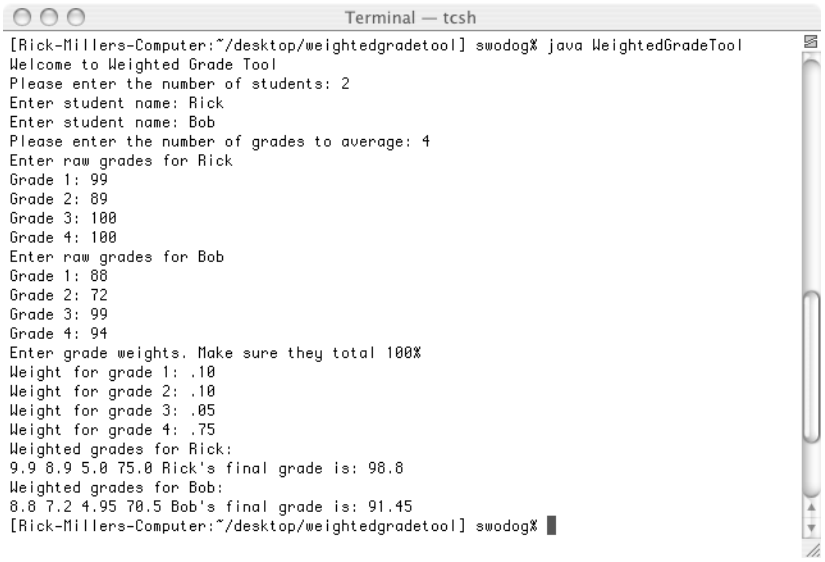


Figure 8-24: Results of Running Example 8.12

The main() method's String Array

Now that you have a better understanding of arrays the main() method's String array will make much more sense. This section explains the purpose and use of the main() method's String array.

Purpose And Use Of The main() Method's String Array

The purpose of the main() method's String array is to enable Java applications to accept and act upon command-line arguments. The javac compiler is an example of a program that takes command-line arguments, the most important of which is the name of the file to compile. The previous chapter also gave several examples of accepting program input via the command line. Now that you are armed with a better understanding of how arrays work you now have the knowledge to write programs that accept and process command-line arguments.

Example 8.13 gives a short program that accepts a line of text as a command-line argument and displays it in lower or upper case depending on the first command-line argument. Figure 8.25 shows the results of running this program.

8.13 CommandLine.java

```
1         public class CommandLine {
2           public static void main(String[] args){
3             StringBuffer sb = null;
4             boolean upper_case = false;
5
6            /********** check for upper case option **************/
7            if(args.length > 0){
8               switch(args[0].charAt(0)){
9                 case '-' : switch(args[0].charAt(1)){
10                               case 'U' :
11                               case 'u' : upper_case = true;
12                                          break;
13                               default:  upper_case = false;
14                            }
15                           break;
16                default: upper_case = false;
17
18              }// end outer switch
19
20              sb = new StringBuffer();   //create StringBuffer object
21
22           /******* process text string *********************/
23              if(upper_case){
```



```
24              for(int i = 1; i < args.length; i++){
25                 sb.append(args[ i] + " ");
26               }//end for
27              System.out.println(sb.toString().toUpperCase());
28             }else {
29                 for(int i = 0; i < args.length; i++){
30                  sb.append(args[ i] + " ");
31               }//end for
32              System.out.println(sb.toString().toLowerCase());
33             }//end if/else
34
35          } else { System.out.println("Usage: CommandLine [ -U | -u] Text string");}
36
37        }//end main
38      }//end class
```

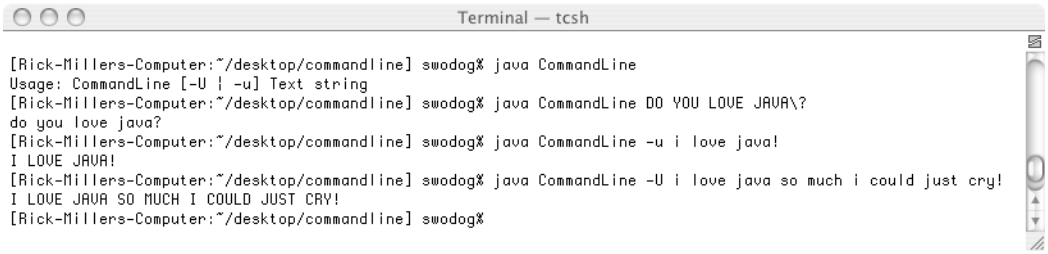


Figure 8-25: Results of Running Example 8.13

Manipulating Arrays With The java.util.Arrays Class

The Java platform makes it easy to perform common array manipulations such as searching and sorting with the java.util.Arrays class. Example 8.14 offers a short program that shows the Arrays class in action sorting an array of integers. Figure 8-26 shows the results of running this program.

8.14 ArraySortApp.java

```
1       import java.util.*;
2
3       public class ArraySortApp {
4         public static void main(String[ ] args){
5          int[ ] int_array = { 100, 45, 9, 1, 34, 22, 6, 4, 3, 2, 99, 66} ;
6
7          for(int i=0; i<int_array.length; i++){
8           System.out.print(int_array[ i] + " ");
9          }
10         System.out.println();
11
12         Arrays.sort(int_array);
13
14         for(int i=0; i<int_array.length; i++){
15          System.out.print(int_array[ i] + " ");
16         }
17         System.out.println();
18        } // end main() method
19      } // end ArraySortApp class definition
```

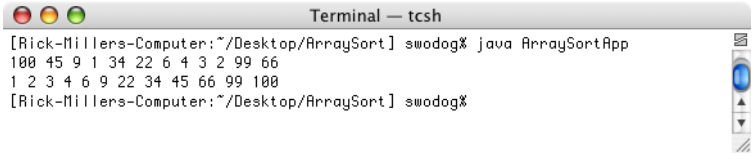


Figure 8-26: Results of Running Example 8.14

Java API Classes Used In This Chapter

Table 8-2 lists and describes the Java API classes and interfaces introduced or utilized in this chapter.

Class	Package	Description
Array	java.lang.reflect	The Array class contains methods that allow you to set and query array elements.
Arrays	java.util	The Arrays class provides the capability to perform common array manipulations like searching and sorting on arrays of primitive and reference types.
BufferedReader	java.io	The BufferedReader class applies buffering to character input streams.
Class	java.lang	This class represents a reference type instance in memory. There is one Class object for each array type loaded into the Java virtual machine.
Cloneable	java.lang	The Cloneable interface indicates that the class that implements it may be cloned.
Double	java.lang	The Double class is a wrapper class for the double primitive type.
InputStream	java.io	The InputStream class is an abstract class that is the superclass of all input streams.
InputStreamReader	java.io	The InputStreamReader class is a character input stream.
Integer	java.lang	The Integer class is a wrapper class for the int primitive type.
IOException	java.io	The IOException is the superclass of all input/output exceptions. It signals a problem with an IO operation.
NumberFormat	java.text	The NumberFormat class in used to format numeric output.
NumberFormatException	java.lang	The NumberFormatExecption signals an illegal number format.
Object	java.lang	The Object class is the root class for all Java and user-defined reference types. This includes Java array types.
Serializable	java.lang	The Serializable interface indicates that the class that implements it may be serialized.
String	java.lang	The String class represents a string of characters and provides methods to manipulate those characters. String objects are immutable, meaning they cannot be changed once created.
StringBuffer	java.lang	The StringBuffer represents a mutable, or changeable, character string.
System	java.lang	The System class provides a platform independent interface to system facilities including properties and input/output streams.

Table 8-2: Java API Classes and Interfaces Referenced in Chapter 8

Summary

Arrays are contiguously allocated memory elements of homogeneous data types. Contiguous means the elements are arranged in memory one after the other. Homogeneous means each element of the array is of the same data type. An array containing n elements is said to have a length equal to n. Array elements are accessed via their index value which ranges from 0 to length - 1. The index value of a particular array element is always one less than the element number you wish to access. (*i.e., 1st element has index 0, 2nd element has index 1, ... , the nth element has index n-1*)

Java array types have special functionality because of their special inheritance hierarchy. Java array types directly subclass the java.lang.Object class and implement the Cloneable and Serializable interfaces. There is also one corresponding java.lang.Class object created in memory for each array-type in the program.

Single-dimensional arrays have one dimension — length. You can get an array's length by accessing its length attribute. Arrays can have elements of primitive or reference types. An array type is created by specifying the type name of array elements followed by one set of brackets [ ]. Use the java.lang.Object and java.lang.Class methods to get information about an array.

Each element of an array is accessed via an index value contained in a set of brackets. Primitive-type element values can be directly assigned to array elements. When an array of primitive types is created each element is initialized to a default value that's appropriate for the element type. Each element of an array of references is initialized to null. Each object that each reference element points to must be dynamically created at some point in the program.

Multidimensional arrays are arrays having more than one dimension. Multidimensional arrays in Java are arrays of arrays. An understanding of single-dimensional Java arrays leads to quick mastery of multidimensional arrays.

To declare a multidimensional array simply add a set of brackets to the array element type for each dimension required in the array. Multidimensional arrays can contain any number of dimensions, however, arrays of two and three dimensions are most common.

Multidimensional arrays can be created using array literal values. The use of array literals enables the initialization of each array element at the point of declaration.

Ragged arrays are multidimensional arrays having final element arrays of various lengths. Ragged arrays can be easily created with array literals. They can also be created by omitting the final dimension at the time of array creation and then dynamically creating each final array with a unique length.

Use the java.util.Arrays class to perform common manipulations such as searching and sorting on arrays of primitive and reference types.

Skill-Building Exercises

1. **Further Research:** Study the Java API classes and interfaces listed in table 8-2 to better familiarize yourself with the functionality they provide.

2. **Further Research:** Conduct a web search and look for different applications for single- and multidimensional arrays.

3. **Single-Dimensional Arrays:** Write a program that lets you create a single-dimensional array of ints of different sizes at runtime using command line inputs. (*Hint: Refer to example 8.9 for some ideas on how to proceed*.) Print the contents of the arrays to the console.

4. **Single-Dimensional Arrays:** Write a program that reverses the order of text entered on the command line. This will require the use of the main() method's String array.

5. **Further Research:** Conduct a web search on different sorting algorithms and how arrays are used to implement these algorithms. Also, several good sources of information regarding sorting algorithms are listed in the references section.

6. **Multidimensional Arrays:** Modify example 8.9 so that it creates two-dimensional arrays of chars. Initialize each element with the character 'c'. Run the program several times to create character arrays of different sizes.

7. **Multidimensional Arrays:** Modify example 8.9 again so that the character array is initialized to the value of the first character read from the command line. (*Hint: Refer to example 8.13 to see how to access the first character of a String*.)

8. **Exploring Arrays Class Functionality:** Research the methods associated with the java.util.Arrays class and note their use. Write a short program that creates arrays of floats, ints, doubles, and Strings and use the various sort methods found in the Arrays class to sort each array. Print the contents of each array to the console both before and after sorting.

Suggested Projects

1. **Matrix Multiplication:** Given two matrices A_{ij} and B_{jk} the product C_{ik} can be calculated with the following equation:

$$C_{ik} = \sum_{j=1}^{n} A_{ij} B_{jk}$$

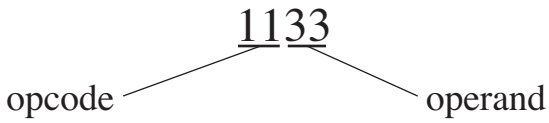
Write a program that multiplies the following matrices together and stores the results in a new matrix. Print the resulting matrix values to the console.

$$\begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$$

2. **Modify Histogram Program:** Modify the Histogram program given in example 8.8 so that it can count the occurrence of the digits 0 through 9 and the punctuation marks period '.', comma ',', question mark '?', colon ':', and semi-colon ';'.

3. **Computer Simulator:** You are a Java developer with a high-tech firm doing contract work for the Department of Defense. You company has won the proposal to develop a proof-of-concept model for an Encrypted Instruction Set Computer System Mark 1. (EISCS Mk1) Your job is to simulate the operation of the EISCS Mk1 with a Java application.

   Supporting Information: The only language a computer understands is its machine language instruction set. The EISCS Mk1 is no different. The EISCS machine language instruction set will consist of a four-digit integer with the two most significant digits being the *operation code* (*opcode*) and the two least significant digits being the *operand*. For example, in the following instruction...



...the number 11 represents the opcode and the number 33 represents the operand. The following table lists and describes each EISCS machine instruction.

Opcode	Mnemonic	Description
Input/Output Operations		
10	READ	Reads an integer value from the console and stores it in memory location identified by the *operand*.
11	WRITE	Writes the integer value stored in memory location *operand* to the console.
Load/Store Operations		
20	LOAD	Loads the integer value stored at memory location *operand* into the accumulator.
21	STORE	Stores the integer value residing in the accumulator into memory location *operand*.
Arithmetic Operations		
30	ADD	Adds the integer value located in memory location *operand* to the value stored in the accumulator and leaves the result in the accumulator.
31	SUB	Subtracts the integer value located in memory location *operand* from the value stored in the accumulator and leaves the result in the accumulator.
32	MUL	Multiplies the integer value located in memory location *operand* by the value stored in the accumulator and leaves the result in the accumulator.
33	DIV	Divides the integer value stored in the accumulator by the value located in memory location *operand*.
Control and Transfer Operations		
40	BRANCH	Unconditional jump to memory location *operand*.
41	BRANCH_NEG	If accumulator value is less than zero jump to memory location *operand*.
42	BRANCH_ZERO	If accumulator value is zero then jump to memory location *operand*.
43	HALT	Stop program execution.

Table 8-3: EISCS Machine Instructions

Sample Program: Using the instruction set given in table 8-3 you can write simple programs that will run on the EISCS Mk1 computer simulator. The following program reads two numbers from the input, multiplies them together, and writes the results to the console.

Memory Location	Instruction / Contents	Action
00	1007	Read integer into memory location 07
01	1008	Read integer into memory location 08
02	2007	Load contents of memory location 07 into accumulator
03	3208	Multiply value located in memory location 08 by value stored in accumulator. Leave result in accumulator

Memory Location	Instruction / Contents	Action
04	2109	Store value currently in accumulator to memory location 09
05	1109	Write the value located in memory location 09 to the console
06	4010	Jump to memory location 10
07		
08		
09		
10	4300	Halt program

Basic Operation: This section discusses several aspects of the EISCS computer simulation operation to assist you in completing the project.

Memory: The machine language instructions that comprise an EISCS program must be loaded into memory before the program can be executed by the simulator. Represent the computer simulator's memory as an array of integers 100 elements long.

Instruction Decoding: Instructions are fetched one at a time from memory and decoded into opcodes and operands before being executed. The following code sample demonstrates one possible decoding scheme:

```
instruction = memory[ program_counter++] ;
operation_code = instruction / 100;
operand = instruction % 100;
```

Hints:

- Use switch/case structure to implement the instruction execution logic
- You may hard code sample programs in your simulator or allow a user to enter a program into memory via the console
- Use an array of 100 integers to represent memory

Self-Test Questions

1. Arrays are contiguously allocated memory elements of homogeneous data types. Explain in your own words what this means.

2. What's the difference between arrays of primitive types vs. arrays of reference types?

3. Java array types directly subclass what Java class?

4. Every Java array type has a corresponding object of this type in memory. What type is it?

5. What two interfaces do Java array types implement?

6. How do you determine the length of an array?

7. Java multidimensional arrays are _____ of _____.

8. When a multidimensional array is created which dimensions are optional and which dimensions are mandatory?

9. What is meant by the term “ragged array”?

10. What’s the purpose of the main() method String array?

References

David Flanagan. *Java In A Nutshell: A Desktop Quick Reference*. O’Reilly and Associates, Inc. Sebastopol, CA. ISBN: 0-596-00283-1

James Gosling, et. al. *The Java™ Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, MA. ISBN: 0-201-31008-2

Jon Meyer, et. al. *Java™ Virtual Machine*. O’Reilly and Associates, Inc. Sebastopol, CA. ISBN: 1-56592-194-1

Rick Miller. *C++ For Artists: The Art, Philosophy, And Science Of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA. ISBN: 1-932504-02-8

Donald E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching,* Second Edition. Addison-Wesley. Reading Massachusetts. ISBN: 0-201-89685-0

Notes

