

8 Arrays

Chapter Objectives

- Define and use arrays for basic data organization.
- Discuss bounds checking and techniques for managing capacity.
- Discuss the issues related to arrays as objects and arrays of objects.
- Explore the use of command-line arguments.
- Describe the syntax and use of variable-length parameter lists.
- Discuss the creation and use of multidimensional arrays.
- Explore polygon and polyline shapes

*In our programming efforts, we often want to organize objects or primitive data in a form that is easy to access and modify. The `ArrayList` class, explored in **Chapter 5** , was used for exactly that purpose. As the class name implies, an `ArrayList` is implemented using arrays, which are programming constructs that group data into lists. In this chapter, we'll explore the details of arrays, which are a fundamental component of most high-level languages. In the Graphics Track sections of this chapter, we explore methods that let us draw complex multisided figures, introduce the choice box control, and see how audio clips can be played in a JavaFX program.*

8.1 Array Elements

An *array* is a simple but powerful programming language construct used to group and organize data. When writing a program that manages a large amount of information, such as a list of 100 names, it is not practical to declare separate variables for each piece of data. Arrays solve this problem by letting us declare one variable that can hold multiple, individually accessible values.

An array is a list of values. Each value is stored at a specific, numbered position in the array. The number corresponding to each position is called an **index** ⓘ or a *subscript*. **Figure 8.1** 📄 shows an array of integers and the indexes that correspond to each position. The array is called `height`; it contains integers that represent several peoples' heights in inches.

	height
0	69
1	61
2	70
3	74
4	62
5	69
6	66
7	73
8	79
9	62
10	70

Figure 8.1 An array called `height` containing integer values


In Java, array indexes always begin at zero. Therefore, the value stored at index 5 is actually the sixth value in the array. The array shown in [Figure 8.1](#) has 11 values, indexed from 0 to 10.

Key Concept

An array of size N is indexed from 0 to $N-1$.

To access a value in an array, we use the name of the array followed by the index in square brackets. For example, the following expression refers to the ninth value in the array `height`:

```
height[8]
```

According to **Figure 8.1** , `height[8]` (pronounced height-sub-eight) contains the value 79. Don't confuse the value of the index, in this case 8, with the value stored in the array at that index, in this case 79.

The expression `height[8]` refers to a single integer stored at a particular memory location. It can be used wherever an integer variable can be used. Therefore, you can assign a value to it, use it in calculations, print its value, and so on. Furthermore, because array indexes are integers, you can use integer expressions to specify the index used to access an array. These concepts are demonstrated in the following lines of code:

```
height[2] = 72;  
height[count] = feet * 12;  
average = (height[0] + height[1] + height[2]) / 3;  
System.out.println("The middle value is " + height[MAX/2]);  
pick = height[rand.nextInt(11)];
```

Arrays are stored contiguously in memory, meaning that the elements are stored one right after the other in memory just as we picture them conceptually. This makes an array extremely efficient in terms of accessing any particular element by its index. Internally, to determine the address of any particular element, the index is multiplied by the


size of each element, and added to the memory address of the starting point of the array. That's why array indexes begin at zero instead of one—to make that computation as easy as possible. So from an efficiency point of view, it's as easy to access the 500th element in the array as it is to access the first element.

Self-Review Questions

(see answers in [Appendix L](#) )


SR 8.1 What is an array?

SR 8.2 How is each element of an array referenced?


SR 8.3 Based on the array shown in [Figure 8.1](#) , what are each of the following?

- a. `height[1]`
- b. `height[2] + height[5]`
- c. `height[2 + 5]`
- d. the value stored at index 8
- e. the fourth value
- f. `height.length`

8.2 Declaring and Using Arrays

In Java, arrays are objects. To create an array, the reference to the array must be declared. The array can then be instantiated using the `new` operator, which allocates memory space to store values. The following code represents the declaration for the array shown in **Figure 8.1** :

```
int[] height = new int[11];
```

The variable `height` is declared to be an array of integers whose type is written as `int[]`. All values stored in an array have the same type (or are at least compatible). For example, we can create an array that can hold integers or an array that can hold strings, but not an array that can hold both integers and strings. An array can be set up to hold any primitive type or any object (class) type. A value stored in an array is sometimes called an *array element*, and the type of values that an array holds is called the **element type**  of the array.

Key Concept

In Java, an array is an object that must be instantiated.

Note that the type of the array variable (`int[]`) does not include the size of the array. The instantiation of `height`, using the `new` operator, reserves the memory space to store 11 integers indexed from 0 to 10. Once an array is declared to be a certain size, the number of values it can hold cannot be changed.



Overview of arrays.

The example shown in [Listing 8.1](#) creates an array called `list` that can hold 15 integers, which it loads with successive increments of 10. It then changes the value of the sixth element in the array (at index 5). Finally, it prints all values stored in the array.

Listing 8.1

```
//*****  
  
//  BasicArray.java      Author: Lewis/Loftus  
//  
//  Demonstrates basic array declaration and use.
```



```

//*****

public class BasicArray
{
    //-----

    //  Creates an array, fills it with various integer values,
    //  modifies one value, then prints them out.
    //-----

    public static void main(String[] args)
    {
        final int LIMIT = 15, MULTIPLE = 10;

        int[] list = new int[LIMIT];

        // Initialize the array values
        for (int index = 0; index < LIMIT; index++)
            list[index] = index * MULTIPLE;


        list[5] = 999; // change one array value

        // Print the array values
        for (int value : list)
            System.out.print(value + " ");
    }
}

```

Output

```
0  10  20  30  40  999  60  70  80  90  100  110  120  130
140
```

Figure 8.2  shows the array as it changes during the execution of the `BasicArray` program. It is often convenient to use `for` loops when handling arrays, because the number of positions in the array is constant. Note that a constant called `LIMIT` is used in several places in the `BasicArray` program. This constant is used to declare the size of the array and to control the `for` loop that initializes the array values.

The array is created with 15 elements, indexed from 0 to 14

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

After three iterations of the first loop

0	0
1	10
2	20
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

After completing the first loop

0	0
1	10
2	20
3	30
4	40
5	50
6	60
7	70
8	80
9	90
10	100
11	110
12	120
13	130
14	140

After changing the value of list[5]

0	0
1	10
2	20
3	30
4	40
5	999
6	60
7	70
8	80
9	90
10	100
11	110
12	120
13	130
14	140

Figure 8.2 The array `list` as it changes in the `BasicArray` program

The iterator version of the `for` loop is used to print the values in the array. Recall from [Chapter 5](#) that this version of the `for` loop extracts each value in the specified iterator. Every Java array is an iterator, so this type of loop can be used whenever we want to process every element stored in an array.

The square brackets used to indicate the index of an array are treated as an operator in Java. Therefore, just like the `+` operator or the `<=` operator, the index operator (`[]`) has a precedence relative to the other Java operators that determines when it is executed. It has the highest precedence of all Java operators.

Bounds Checking


Java performs *automatic bounds checking*, which ensures that the index is in range for the array being referenced. Whenever a reference to an array element is made, the index must be greater than or equal to zero and less than the size of the array. For example, suppose an array called `prices` is created with 25 elements. The valid indexes for the array are from 0 to 24. Whenever a reference is made to a particular element in the array (such as `prices[count]`), the value of the index is checked. If it is in the valid range of indexes for the array (0 to 24), the reference is carried out. If the index is not valid, an exception called `ArrayIndexOutOfBoundsException` is thrown.

Key Concept

Bounds checking ensures that an index used to refer to an array element is in range.

Of course, in our programs we'll want to perform our own bounds checking. That is, we'll want to be careful to remain within the bounds of the array and process every element we intend to. Because array indexes begin at zero and go up to one less than the size of the array, it is easy to create *off-by-one errors* in a program, which are problems created by processing all but one element or by attempting to index one element too many.

One way to check for the bounds of an array is to use the `length` constant, which is held in the array object and stores the size of the array. It is a public constant and therefore can be referenced directly. For example, after the array `prices` is created with 25 elements, the constant `prices.length` contains the value 25. Its value is set once when the array is first created and cannot be changed. The `length` constant, which is an integral part of each array, can be used when the array size is needed without having to create a separate constant. Remember that the length of the array is the number of elements it can hold, thus the maximum index of an array is `length-1`.

Let's look at another example. The program shown in [Listing 8.2](#)  reads 10 integers into an array called `numbers`, and then prints them in reverse order.

Listing 8.2

```
//*****
```

```

// ReverseOrder.java          Author: Lewis/Loftus
//
// Demonstrates array index processing.
//*****

import java.util.Scanner;

public class ReverseOrder
{
    //-----

    // Reads a list of numbers from the user, storing them in
    an
    // array, then prints them in the opposite order.
    //-----

    -----

    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);

        double[] numbers = new double[10];

        System.out.println("The size of the array: " +
        numbers.length);

        for (int index = 0; index < numbers.length; index++)
        {

```

```

        System.out.print("Enter number " + (index+1) + ": ");

        numbers[index] = scan.nextDouble();

    }

    System.out.println("The numbers in reverse order:");

    for (int index = numbers.length-1; index >= 0; index--)
        System.out.print(numbers[index] + " ");

    }

}

```

Output

```

|
The size of the array: 10
Enter number 1: 18.36
Enter number 2: 48.9
Enter number 3: 53.5
Enter number 4: 29.06
Enter number 5: 72.404
Enter number 6: 34.8
Enter number 7: 63.41
Enter number 8: 45.55
Enter number 9: 69.0
Enter number 10: 99.18
The numbers in reverse order:
99.18  69.0  45.55  63.41  34.8  72.404  29.06  53.5  48.9

```

Note that in the `ReverseOrder` program, the array `numbers` is declared to have 10 elements and therefore is indexed from 0 to 9. The index range is controlled in the `for` loops by using the `length` field of the array object. You should carefully set the initial value of loop control variables and the conditions that terminate loops to guarantee that all intended elements are processed and only valid indexes are used to reference an array element.

The `LetterCount` example, shown in [Listing 8.3](#), uses two arrays and a `String` object. The array called `upper` is used to store the number of times each uppercase alphabetic letter is found in the string. The array called `lower` serves the same purpose for lowercase letters.

Listing 8.3

```
//*****

//  LetterCount.java      Author: Lewis/Loftus
//
//  Demonstrates the relationship between arrays and strings.
//*****

import java.util.Scanner;
```



```
public class LetterCount
{
    //-----
    // Reads a sentence from the user and counts the number of
    // uppercase and lowercase letters contained in it.
    //-----
    public static void main(String[] args)
    {
        final int NUMCHARS = 26;

        Scanner scan = new Scanner(System.in);

        int[] upper = new int[NUMCHARS];
        int[] lower = new int[NUMCHARS];

        char current;    // the current character being
        processed

        int other = 0;    // counter for non-alphabets

        System.out.println("Enter a sentence:");
        String line = scan.nextLine();

        //Count the number of each letter occurrence
        for (int ch = 0; ch < line.length(); ch++)
        {
            current = line.charAt(ch);
```

```

        if (current >= 'A' && current <= 'Z')
            upper[current-'A']++;
        else
            if (current >= 'a' && current <= 'z')
                lower[current-'a']++;
            else
                other++;
    }

    //Print the results
    System.out.println();
    for (int letter=0; letter < upper.length; letter++)
    {
        System.out.print((char) (letter + 'A'));
        System.out.print(": " + upper[letter]);
        System.out.print("\t\t" + (char) (letter + 'a'));
        System.out.println(": " + lower[letter]);
    }

    System.out.println();
    System.out.println("Non-alphabetic characters: " +
other);
}
}

```

Output

Enter a sentence:

In Casablanca, Humphrey Bogart never says "Play it again,
Sam."

A: 0 a: 10

B: 1 b: 1

C: 1 c: 1

D: 0 d: 0

E: 0 e: 3

F: 0 f: 0

G: 0 g: 2

H: 1 h: 1

I: 1 i: 2

J: 0 j: 0

K: 0 k: 0

L: 0 l: 2

M: 0 m: 2

N: 0 n: 4

O: 0 o: 1

P: 1 p: 1

Q: 0 q: 0

R: 0 r: 3

S: 1 s: 3

T: 0 t: 2

U: 0 u: 1

V: 0 v: 1

```
W: 0      w: 0
X: 0      x: 0
Y: 0      y: 3
Z: 0      z: 0

Non-alphabetic characters: 14
```

Because there are 26 letters in the English alphabet, both the `upper` and `lower` arrays are declared with 26 elements. Each element contains an integer that is initially zero by default. The `for` loop scans through the string one character at a time. The appropriate counter in the appropriate array is incremented for each character found in the string.




Discussion of the `LetterCount` example.

Both the counter arrays are indexed from 0 to 25. We have to map each character to a counter. A logical way to do this is to use `upper[0]` to count the number of `'A'` characters found, `upper[1]` to count the number of `'B'` characters found, and so on. Likewise, `lower[0]` is used to count `'a'` characters, `lower[1]` is used to count

'b' characters, and so on. A separate variable called `other` is used to count any nonalphabetic characters that are encountered.

Note that to determine if a character is an uppercase letter we used the boolean expression `(current >= 'A' && current <= 'Z')`. A similar expression is used for determining the lowercase letters. We could have used the static methods `isUpperCase` and `isLowerCase` in the `Character` class to make these determinations but didn't in this example to drive home the point that because characters are based on the Unicode character set, they have a specific numeric value and order that we can use in our programming.

We use the current character to calculate which index in the array to reference. We have to be careful when calculating an index to ensure that it remains within the bounds of the array and matches to the correct element. Remember that in the Unicode character set, the uppercase and lowercase alphabetic letters are continuous and in order (see [Appendix C](#) ). Therefore, taking the numeric value of an uppercase letter such as `'E'` (which is 69) and subtracting the numeric value of the character `'A'` (which is 65) yields 4, which is the correct index for the counter of the character `'E'`. Note that nowhere in the program do we actually need to know the specific numeric values for each letter.

Alternate Array Syntax

Syntactically, there are two ways to declare an array reference in Java. The first technique, which is used in the previous examples and throughout this text, is to associate the brackets with the type of values stored in the array. The second technique is to associate the brackets with the name of the array. Therefore, the following two declarations are equivalent:

```
int[] grades;  
int grades[];
```

Although there is no difference between these declaration techniques as far as the compiler is concerned, the first is consistent with other types of declarations. The declared type is explicit if the array brackets are associated with the element type, especially if there are multiple variables declared on the same line. Therefore, we associate the brackets with the element type throughout this text.

Initializer Lists

You can use an *initializer list* to instantiate an array and provide the initial values for the elements of the array. It is essentially the same idea as initializing a variable of a primitive data type in its declaration except that an array requires several values.



Key Concept

An initializer list can be used to instantiate an array object instead of using the `new` operator.

The items in an initializer list are separated by commas and delimited by braces (`{}`). When an initializer list is used, the `new` operator is not used. The size of the array is determined by the number of items in the initializer list. For example, the following declaration instantiates the array `scores` as an array of eight integers, indexed from 0 to 7 with the specified initial values:

```
int[] scores = {87, 98, 69, 87, 65, 76, 99, 83};
```

An initializer list can be used only when an array is first declared.

The type of each value in an initializer list must match the type of the array elements. Let's look at another example:

```
char[] vowels = {'A', 'E', 'I', 'O', 'U'};
```

In this case, the variable `vowels` is declared to be an array of five characters, and the initializer list contains character literals.

The program shown in [Listing 8.4](#) demonstrates the use of an initializer list to instantiate an array.

Listing 8.4

```
//*****

//  Primes.java      Author: Lewis/Loftus
//
//  Demonstrates the use of an initializer list for an array.
//*****

public class Primes
{
    //-----
    //  Stores some prime numbers in an array and prints them.
    //-----

    public static void main(String[] args)
    {
        int[] primeNums = {2, 3, 5, 7, 11, 13, 17, 19};

        System.out.println("Array length: " + primeNums.length);

        System.out.println("The first few prime numbers are:");
```




```
        for (int prime : primeNums)
            System.out.print(prime + " ");
    }
}
```

Output

```
|
Array length: 8
The first few prime numbers are:
2  3  5  7  11  13  17  19
```

Arrays as Parameters

An entire array can be passed as a parameter to a method. Because an array is an object, when an entire array is passed as a parameter, a copy of the reference to the original array is passed. We discussed this issue as it applies to all objects in [Chapter 7](#) .

Key Concept

An entire array can be passed as a parameter, making the formal parameter an alias of the

original.

A method that receives an array as a parameter can permanently change an element of the array, because it is referring to the original element value. The method cannot permanently change the reference to the array itself, because a copy of the original reference is sent to the method. These rules are consistent with the rules that govern any object type.

An element of an array can be passed to a method as well. If the element type is a primitive type, a copy of the value is passed. If that element is a reference to an object, a copy of the object reference is passed. As always, the impact of changes made to a parameter inside the method depends on the type of the parameter. We discuss arrays of objects further in the next section.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 8.4 What is an array's element type?

SR 8.5 Describe the process of creating an array. When is memory allocated for the array?

SR 8.6 Write an array declaration to represent the ages of all 100 children attending a summer camp.

SR 8.7 Write an array declaration to represent the counts of how many times each face appeared when a standard six-sided die is rolled.

SR 8.8 Explain the concept of array bounds checking. What happens when a Java array is indexed with an invalid value?

SR 8.9 What is an off-by-one error? How does it relate to arrays?

SR 8.10 Write code that increments (by one) each element of an array of integers named `values`.

SR 8.11 Write code that computes and prints the sum of the elements of an array of integers named `values`.

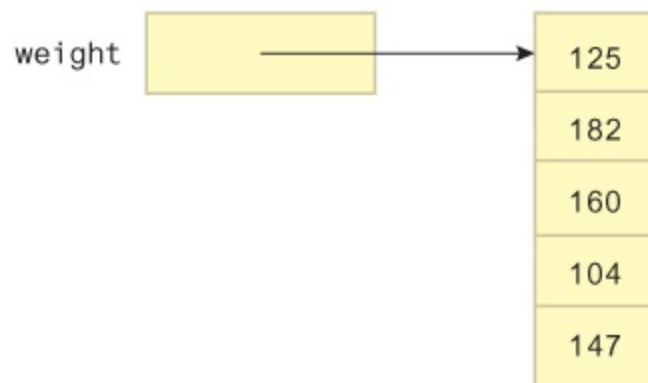
SR 8.12 What does an array initializer list accomplish?

SR 8.13 Can an entire array be passed as a parameter? How is this accomplished?

8.3 Arrays of Objects

In the previous examples in this chapter, we used arrays to store primitive types such as integers and characters. Arrays can also store references to objects as elements. Fairly complex information management structures can be created using only arrays and other objects. For example, an array could contain objects, and each of those objects could consist of several variables and the methods that use them. Those variables could themselves be arrays, and so on. The design of a program should capitalize on the ability to combine these constructs to create the most appropriate representation for the information.

Keep in mind that the array itself is an object. So it would be appropriate to picture an array of `int` values called `weight` as follows:



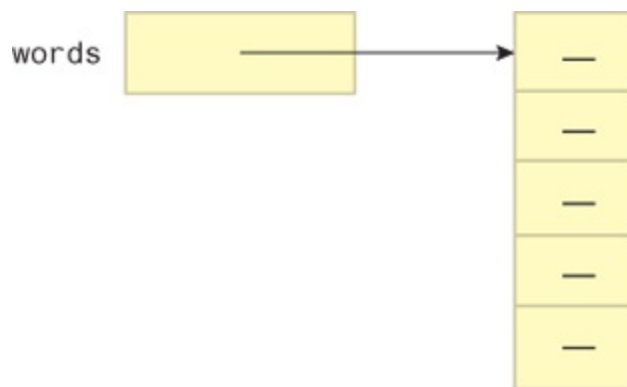
Key Concept

Instantiating an array of objects reserves room to store references only. The objects that are stored in each element must be instantiated separately.

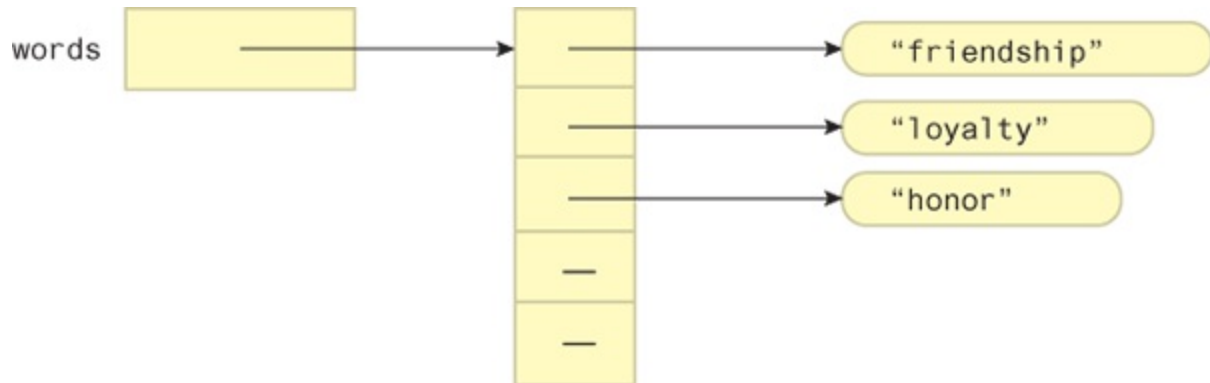
When we store objects in an array, each element is a separate object. That is, an array of objects is really an array of object references. Consider the following declaration:

```
String[] words = new String[5];
```

The variable `words` is an array of references to `String` objects. The `new` operator in the declaration instantiates the array and reserves space for five `String` references. This declaration does not create any `String` objects; it merely creates an array that holds references to `String` objects. Initially, the array looks like this:



After a few `String` objects are created and put in the array, it might look like this:



The `words` array is an object, and each character string it holds is its own object. Each object contained in an array has to be instantiated separately.

Keep in mind that `String` objects can be represented as string literals. So the following declaration creates an array called `verbs` and uses an initializer list to populate it with several `String` objects, each instantiated using a string literal:

```
String[] verbs = {"play", "work", "eat", "sleep"};
```

The program called `GradeRange` shown in [Listing 8.5](#) creates an array of `Grade` objects, then prints them. The `Grade` objects are created using several `new` operators in the initialization list of the array.

Listing 8.5

```
//*****

//  GradeRange.java      Author: Lewis/Loftus
//
//  Demonstrates the use of an array of objects.
//*****

public class GradeRange
{
    //-----

    //  Creates an array of Grade objects and prints them.
    //-----

    public static void main(String[] args)
    {
        Grade[] grades =
        {
            new Grade("A", 95), new Grade("A-", 90),
            new Grade("B+", 87), new Grade("B", 85), new
Grade("B-", 80),
            new Grade("C+", 77), new Grade("C", 75), new
Grade("C-", 70),
            new Grade("D+", 67), new Grade("D", 65), new
Grade("D-", 60),
```

```

        new Grade("F", 0)

    };

    for (Grade letterGrade : grades)
        System.out.println(letterGrade);
    }
}

```

Output

```

|
A      95
A-     90
B+     87
B      85
B-     80
C+     77
C      75
C-     70
D+     67
D      65
D-     60
F       0

```

The `Grade` class is shown in [Listing 8.6](#). Each `Grade` object represents a letter grade for a school course and includes a numerical

lower bound. The values for the grade name and lower bound can be set using the `Grade` constructor, or using appropriate mutator methods. Accessor methods are also defined, as is a `toString` method to return a string representation of the grade. The `toString` method is automatically invoked when the grades are printed in the `main` method.

Listing 8.6

```
//*****  
  
//  Grade.java      Author: Lewis/Loftus  
//  
//  Represents a school grade.  
//*****  
  
  
public class Grade  
{  
    private String name;  
    private int lowerBound;  
  
    //-----  
    -----  
    // Constructor: Sets up this Grade object with the  
    specified  
    // grade name and numeric lower bound.
```

```
//-----  
-----  
public Grade(String grade, int cutoff)  
{  
    name = grade;  
    lowerBound = cutoff;  
}
```

```
//-----  
-----  
// Returns a string representation of this grade.  
//-----  
-----
```

```
public String toString()  
{  
    return name + "\t" + lowerBound;  
}
```

```
//-----  
-----  
// Name mutator.  
//-----  
-----
```

```
public void setName(String grade)  
{  
    name = grade;  
}
```

```
//-----
```

```
// Lower bound mutator.
```

```
//-----
```

```
public void setLowerBound(int cutoff)
```

```
{
```

```
    lowerBound = cutoff;
```

```
}
```

```
//-----
```

```
// Name accessor.
```

```
//-----
```

```
public String getName()
```

```
{
```

```
    return name;
```

```
}
```

```
//-----
```

```
// Lower bound accessor.
```

```
//-----
```


```
public int getLowerBound()
```

```
{
```

```
    return lowerBound;
```

```
}
```

```
}
```

Let's look at another example. **Listing 8.7**  shows the `Movies` class, which contains a `main` method that creates, modifies, and examines a DVD collection.

Listing 8.7

```
//*****

//  Movies.java      Author: Lewis/Loftus
//
//  Demonstrates the use of an array of objects.
//*****

public class Movies
{
    //-----

    // Creates a DVDCollection object and adds some DVDs to it.
    Prints
    // reports on the status of the collection.
    //-----

    public static void main(String[] args)
    {
```

```
DVDCollection movies = new DVDCollection();

movies.addDVD("The Godfather", "Francis Ford Coppola",
1972, 24.95, true);

movies.addDVD("District 9", "Neill Blomkamp", 2009,
19.95, false);

movies.addDVD("Iron Man", "Jon Favreau", 2008, 15.95,
false);

movies.addDVD("All About Eve", "Joseph Mankiewicz",
1950, 17.50, false);

movies.addDVD("The Matrix", "Andy & Lana Wachowski",
1999, 19.95, true);

System.out.println(movies);

movies.addDVD("Iron Man 2", "Jon Favreau", 2010, 22.99,
false);

movies.addDVD("Casablanca", "Michael Curtiz", 1942,
19.95, false);

System.out.println(movies);
}
}
```

Output

|

~~~~~

My DVD Collection

Number of DVDs: 5

Total cost: \$98.30

Average cost: \$19.66

DVD List:

|         |      |               |                      |
|---------|------|---------------|----------------------|
| \$24.95 | 1972 | The Godfather | Francis Ford Coppola |
|---------|------|---------------|----------------------|

Blu-ray

|         |      |            |                |
|---------|------|------------|----------------|
| \$19.95 | 2009 | District 9 | Neill Blomkamp |
|---------|------|------------|----------------|

|         |      |          |             |
|---------|------|----------|-------------|
| \$15.95 | 2008 | Iron Man | Jon Favreau |
|---------|------|----------|-------------|

|         |      |               |                   |
|---------|------|---------------|-------------------|
| \$17.50 | 1950 | All About Eve | Joseph Mankiewicz |
|---------|------|---------------|-------------------|

|         |      |            |                       |
|---------|------|------------|-----------------------|
| \$19.95 | 1999 | The Matrix | Andy & Lana Wachowski |
|---------|------|------------|-----------------------|

Blu-ray

~~~~~

My DVD Collection

Number of DVDs: 7

Total cost: \$141.24

Average cost: \$20.18

DVD List:

\$24.95	1972	The Godfather	Francis Ford Coppola
---------	------	---------------	----------------------


```

Blu-ray
$19.95  2009      District 9      Neill Blomkamp
$15.95  2008      Iron Man        Jon Favreau
$17.50  1950      All About Eve   Joseph Mankiewicz
$19.95  1999      The Matrix      Andy & Lana Wachowski

Blu-ray
$22.99  2010      Iron Man 2      Jon Favreau
$19.95  1942      Casablanca      Michael Curtiz

```

Each DVD added to the collection is specified by its title, director, year of release, purchase price, and whether or not it is in Blu-ray format.

Listing 8.8  shows the `DVDCollection` class. It contains an array of `DVD` objects representing the collection. It maintains a count of the DVDs in the collection and their combined value. It also keeps track of the current size of the collection array so that a larger array can be created if too many DVDs are added to the collection.

Listing 8.8

```

//*****

//  DVDCollection.java      Author: Lewis/Loftus
//
//  Represents a collection of DVD movies.
//*****

```

```
import java.text.NumberFormat;

public class DVDCollection
{
    private DVD[] collection;
    private int count;
    private double totalCost;

    //-----

    // Constructor: Creates an initially empty collection.
    //-----

    public DVDCollection()
    {
        collection = new DVD[100];
        count = 0;
        totalCost = 0.0;
    }

    //-----

    // Adds a DVD to the collection, increasing the size of the
    // collection array if necessary.
    //-----

    public void addDVD(String title, String director, int year,
```



```

        double cost, boolean bluray)
    {
        if (count == collection.length)
            increaseSize();

        collection[count] = new DVD(title, director, year, cost,
bluray);
        totalCost += cost;
        count++;
    }

    //-----

    // Returns a report describing the DVD collection.
    //-----

    public String toString()
    {
        NumberFormat fmt = NumberFormat.getCurrencyInstance();

        String report =
"~~~~~\n";
        report += "My DVD Collection\n\n";

        report += "Number of DVDs: " + count + "\n";
        report += "Total cost: " + fmt.format(totalCost) + "\n";
        report += "Average cost: " +
fmt.format(totalCost/count);
    }
}

```

```
report += "\n\nDVD List:\n\n";
```

```
for (int dvd = 0; dvd < count; dvd++)
```

```
    report += collection[dvd].toString() + "\n";
```

```
return report;
```

```
}
```

```
//-----  
-----  
// Increases the capacity of the collection by creating a  
// larger array and copying the existing collection into  
it.
```

```
//-----  
-----
```

```
private void increaseSize()
```

```
{
```

```
    DVD[] temp = new DVD[collection.length * 2];
```

```
    for (int dvd = 0; dvd < collection.length; dvd++)
```

```
        temp[dvd] = collection[dvd];
```


```
    collection = temp;
```

```
}
```

```
}
```

The `collection` array is instantiated in the `DVDCollection` constructor. Every time a DVD is added to the collection (using the `addDVD` method), a new `DVD` object is created and a reference to it is stored in the `collection` array.

Each time a DVD is added to the collection, we check to see whether we have reached the current capacity of the `collection` array. If we didn't perform this check, an exception would eventually be thrown when we try to store a new `DVD` object at an invalid index. If the current capacity has been reached, the private `increaseSize` method is invoked, which first creates an array that is twice as big as the current `collection` array. Each DVD in the existing collection is then copied into the new array. Finally, the `collection` reference is set to the larger array. Using this technique, we theoretically never run out of room in our DVD collection. The user of the `DVDCollection` object (the `main` method in this case) never has to worry about running out of space, because it's all handled internally.

Figure 8.3  shows a UML class diagram of the `Movies` program. Recall that the open diamond indicates aggregation. The cardinality of the relationship is also noted: a `DVDCollection` object contains zero or more `DVD` objects.

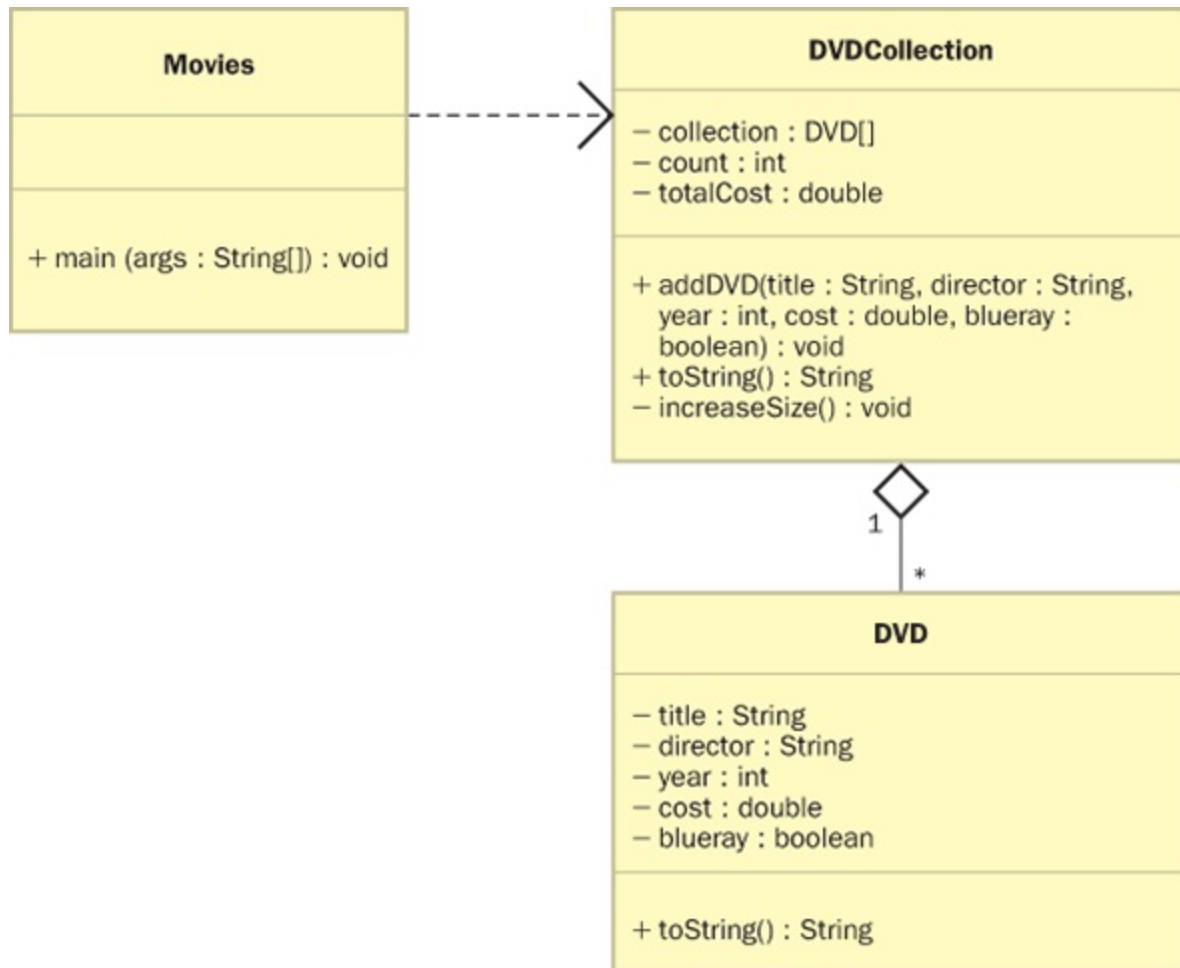



Figure 8.3 A UML class diagram of the `Movies` program

The `toString` method of the `DVDCollection` class returns an entire report summarizing the collection. The report is created, in part, using calls to the `toString` method of each `DVD` object stored in the collection. [Listing 8.9](#)  shows the `DVD` class.

Listing 8.9

```
//*****
```

```

// DVD.java      Author: Lewis/Loftus

//
// Represents a DVD video disc.
//*****

import java.text.NumberFormat;

public class DVD
{
    private String title, director;
    private int year;
    private double cost;
    private boolean bluray;

    //-----
    // Creates a new DVD with the specified information.
    //-----

    public DVD(String title, String director, int year, double
cost,
                boolean bluray)
    {
        this.title = title;
        this.director = director;
        this.year = year;
        this.cost = cost;

```

```

        this.bluray = bluray;
    }

    //-----

    // Returns a string description of this DVD.
    //-----

    public String toString()
    {
        NumberFormat fmt = NumberFormat.getCurrencyInstance();

        String description;

        description = fmt.format(cost) + "\t" + year + "\t";
        description += title + "\t" + director;
        if (bluray)
            description += "\t" + "Blu-ray";

        return description;
    }
}

```

Self-Review Questions

(see answers in [Appendix L](#) )

SR 8.14 How is an array of objects created?

SR 8.15 Suppose `team` is an array of strings meant to hold the names of the six players on a volleyball team: Amanda, Clare, Emily, Julie, Katie, and Maria.

- a. Write an array declaration for `team`.
- b. Show how to both declare and populate `team` using an initializer list.

SR 8.16 Assume `Book` is a class whose objects represent books. Assume a constructor of the `Book` class accepts two parameters—the name of the book and the number of pages.

- a. Write a declaration for a variable named `library` that is an array of ten books.
- b. Write a `new` statement that sets the first book in the `library` array to `"Starship Troopers"`, which is 208 pages long.


8.4 Command-Line Arguments

The parameter to the `main` method of a Java application is always an array of `String` objects. We've ignored that parameter in previous examples, but now we can discuss how it might occasionally be useful.

Key Concept

Command-line arguments are stored in an array of `String` objects and are passed to the `main` method.

The Java run-time environment invokes the `main` method when an application is submitted to the interpreter. The `String[]` parameter, which we typically call `args`, represents *command-line arguments* that are provided when the interpreter is invoked. Any extra information on the command line when the interpreter is invoked is stored in the `args` array for use by the program. This technique is another way to provide input to a program.

The program shown in [Listing 8.10](#)  uses command-line arguments to print a nametag. It assumes the first argument represents some type of greeting and the second argument represents a person's name.

Listing 8.10

```
//*****  
  
//  NameTag.java      Author: Lewis/Loftus  
//  
//  Demonstrates the use of command line arguments.  
//*****  
  
  
  
public class NameTag  
{  
  
    //-----  
    -----  
    // Prints a simple name tag using a greeting and a name  
    that is  
    // specified by the user.  
    //-----  
    -----  
  
    public static void main(String[] args)  
    {  
  
        System.out.println();  
        System.out.println("      " + args[0]);  
    }  
}
```

```
        System.out.println("My name is " + args[1]);  
    }  
}
```

Output

```
> java NameTag Howdy John
```

```
Howdy
```

```
My name is John
```

```
> java NameTag Hello Bill
```

```
Hello
```

```
My name is Bill
```

If two strings are not provided on the command line for the `NameTag` program, the `args` array will not contain enough (if any) elements, and the references in the program will cause an `ArrayIndexOutOfBoundsException` to be thrown. If extra information is included on the command line, it will be stored in the `args` array but ignored by the program.

Remember that the parameter to the `main` method is always an array of `String` objects. If you want numeric information to be input as a command-line argument, the program has to convert it from its string representation.

You also should be aware that in some program development environments, a command line is not used to submit a program to the interpreter. In such situations, the command-line information can be specified in some other way. Consult the documentation for these specifics if necessary.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 8.17 What is a command-line argument?

SR 8.18 Write a `main` method for a program that outputs the sum of the string lengths of its first two command-line arguments.

SR 8.19 Write a `main` method for a program that outputs the sum of the values of its first two command-line arguments, which are integers.

8.5 Variable Length Parameter Lists

Suppose we wanted to design a method that processed a different amount of data from one invocation to the next. For example, let's design a method called `average` that accepts a few integer values and returns their average. In one invocation of the method, we might pass in three integers to `average`:

```
mean1 = average(42, 69, 37);
```

In another invocation of the same method, we might pass in seven integers to `average`:

```
mean2 = average(35, 43, 93, 23, 40, 21, 75);
```

Key Concept

A Java method can be defined to accept a varying number of parameters.

To accomplish this, we could define overloaded versions of the `average` method, but that would require that we know the maximum number of parameters there might be and create a separate version of the method for each possibility. Alternatively, we could define the method to accept an array of integers, which could be of different sizes for each call. But that would require packaging the integers into an array in the calling method and passing in one parameter.

Java provides a way to define methods that accept variable-length parameter lists. By using some special syntax in the formal parameter list of the method, we can define the method to accept any number of parameters. The parameters are automatically put into an array for easy processing in the method. For example, the `average` method could be written as follows:

```
public double average(int ... list)
{
    double result = 0.0;

    if (list.length != 0)
    {
        int sum = 0;
        for (int num : list)
            sum += num;
        result = (double)sum / list.length;
    }
}
```

```
    return result;
}
```

Note the way the formal parameters are defined. The ellipsis (three periods in a row) indicates that the method accepts a variable number of parameters. In this case, the method accepts any number of `int` parameters, which it automatically puts into an array called `list`. In the method, we process the array normally.

We can now pass any number of `int` parameters to the `average` method, including none at all. That's why we check to see if the length of the array is zero before we compute the average.

The type of the multiple parameters can be any primitive or object type. For example, the following method accepts and prints multiple `Grade` objects (we defined the `Grade` class earlier in this chapter):


```
public void printGrades(Grade ... grades)
{
    for (Grade letterGrade : grades)
        System.out.println(letterGrade);
}
```

A method that accepts a variable number of parameters can also accept other parameters. For example, the following method accepts

an `int`, a `String` object, and then a variable number of `double` values that will be stored in an array called `nums`:

```
public void test(int count, String name, double ... nums)
{
    // whatever
}
```

The varying parameters must come last in the formal arguments. A single method cannot accept two sets of varying parameters.

Constructors can also be set up to accept a varying number of parameters. The program shown in [Listing 8.11](#)  creates two `Family` objects, passing a varying number of strings (representing the family member names) into the `Family` constructor.

Listing 8.11

```
//*****

//  VariableParameters.java      Author: Lewis/Loftus
//
//  Demonstrates the use of a variable length parameter list.
//*****

```

```
public class VariableParameters
{
    //-----

    // Creates two Family objects using a constructor that
    accepts
    // a variable number of String objects as parameters.
    //-----


    public static void main(String[] args)
    {
        Family lewis = new Family("John", "Sharon", "Justin",
        "Kayla",
        "Nathan", "Samantha");

        Family camden = new Family("Stephen", "Annie", "Matt",
        "Mary",
        "Simon", "Lucy", "Ruthie", "Sam", "David");

        System.out.println(lewis);
        System.out.println();
        System.out.println(camden);
    }
}
```

Output


```
|  
John  
Sharon  
Justin  
Kayla  
Nathan  
Samantha  
  
Stephen  
Annie  
Matt  
Mary  
Simon  
Lucy  
Ruthie  
Sam  
David
```

The `Family` class is shown in [Listing 8.12](#) . The constructor simply stores a reference to the array parameter until it is needed. By using a variable-length parameter list for the constructor, we make it easy to create a family of any size.

Listing 8.12

```
//*****
```

```

// Family.java      Author: Lewis/Loftus
//
// Demonstrates the use of variable length parameter lists.
//*****

public class Family
{
    private String[] members;

    //-----
    // Constructor: Sets up this family by storing the
    (possibly
    // multiple) names that are passed in as parameters.
    //-----
    -----
    public Family(String ... names)
    {
        members = names;
    }

    //-----
    -----
    // Returns a string representation of this family.
    //-----
    -----
    public String toString()

```

```
{  
    String result = "";  
  
    for (String name : members)  
        result += name + "\n";  
  
    return result;  
}  
}
```

Self-Review Questions

(see answers in [Appendix L](#) )

SR 8.20 How can Java methods have variable-length parameter lists?

SR 8.21 Write a method called `distance` that accepts a multiple number of integer parameters (each of which represents the distance of one leg of a journey) and returns the total distance of the trip.

SR 8.22 Write a method called `travelTime` that accepts an integer parameter indicating average speed followed by a multiple number of integer parameters (each of which represents the distance of one leg of a journey) and returns the total time of the trip.

8.6 Two-Dimensional Arrays

The arrays we've examined so far have all been *one-dimensional arrays* in the sense that they represent a simple list of values. As the name implies, a **two-dimensional array** ⓘ has values in two dimensions, which are often thought of as the rows and columns of a table. **Figure 8.4** ☐ graphically compares a one-dimensional array with a two-dimensional array. We must use two indexes to refer to a value in a two-dimensional array, one specifying the row and another the column.

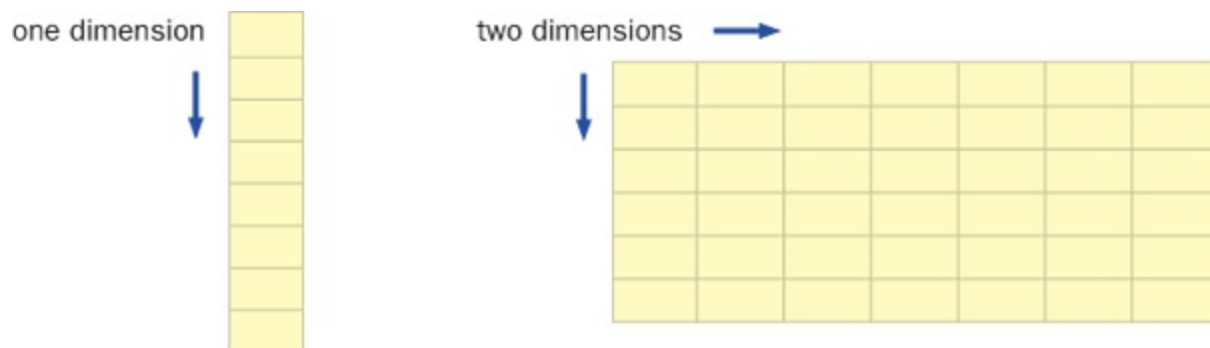



Figure 8.4 A one-dimensional array and a two-dimensional array

Brackets are used to represent each dimension in the array.

Therefore, the type of a two-dimensional array that stores integers is `int[][]`. Technically, Java represents two-dimensional arrays as an array of arrays. A two-dimensional integer array is really a one-dimensional array of references to one-dimensional integer arrays.

The `TwoDArray` program shown in [Listing 8.13](#)  instantiates a two-dimensional array of integers. As with one-dimensional arrays, the size of the dimensions is specified when the array is created. The size of the dimensions can be different.

Listing 8.13

```
//*****

//  TwoDArray.java      Author: Lewis/Loftus
//
//  Demonstrates the use of a two-dimensional array.
//*****

public class TwoDArray
{
    //-----
    -----
    //  Creates a 2D array of integers, fills it with
    increasing
    //  integer values, then prints them out.
    //-----
    -----
    public static void main(String[] args)
    {
        int[][] table = new int[5][10];
```

```

        // Load the table with values
        for (int row = 0; row < table.length; row++)
            for (int col = 0; col < table[row].length; col++)
                table[row][col] = row * 10 + col;

        // Print the table
        for (int row = 0; row < table.length; row++)
        {
            for (int col = 0; col < table[row].length; col++)
                System.out.print(table[row][col] + "\t");
            System.out.println();
        }
    }
}

```

Output

0	1	2	3	4	5	6	7
8	9						
10	11	12	13	14	15	16	17
18	19						
20	21	22	23	24	25	26	27
28	29						
30	31	32	33	34	35	36	37
38	39						


```

public class SodaSurvey
{
    //-----

    // Determines and prints the average of each row (soda)
    and each
    // column (respondent) of the survey scores.
    //-----

    public static void main(String[] args)
    {
        int[][] scores = { {3, 4, 5, 2, 1, 4, 3, 2, 4, 4},
                           {2, 4, 3, 4, 3, 3, 2, 1, 2, 2},
                           {3, 5, 4, 5, 5, 3, 2, 5, 5, 5},
                           {1, 1, 1, 3, 1, 2, 1, 3, 2, 4} };

        final int SODAS = scores.length;
        final int PEOPLE = scores[0].length;

        int[] sodaSum = new int[SODAS];
        int[] personSum = new int[PEOPLE];

        for (int soda = 0; soda < SODAS; soda++)
            for (int person = 0; person < PEOPLE; person++)
            {
                sodaSum[soda] += scores[soda][person];
                personSum[person] += scores[soda][person];
            }
    }
}

```



```

    }

    DecimalFormat fmt = new DecimalFormat("0.##");

    System.out.println("Averages:\n");

    for (int soda = 0; soda < SODAS; soda++)

        System.out.println("Soda #" + (soda+1) + ": " +

            fmt.format((float)sodaSum[soda]/PEOPLE));

    System.out.println();

    for (int person = 0; person < PEOPLE; person++)

        System.out.println("Person #" + (person+1) + ": " +

            fmt.format((float)personSum[person]/SODAS));

    }

}

```

Output

```

|
Averages:

Soda #1: 3.2
Soda #2: 2.6
Soda #3: 4.2
Soda #4: 1.9

```

```
Person #1: 2.2
Person #2: 3.5
Person #3: 3.2
Person #4: 3.5
Person #5: 2.5
Person #6: 3
Person #7: 2
Person #8: 2.8
Person #9: 3.2
Person #10: 3.8
```

Suppose a soda manufacturer held a taste test for four new flavors to see how people liked them. The manufacturer got 10 people to try each new flavor and give it a score from 1 to 5, where 1 equals poor and 5 equals excellent. The two-dimensional array called `scores` in the `SodaSurvey` program stores the results of that survey. Each row corresponds to a soda and each column in that row corresponds to the person who tasted it. More generally, each row holds the responses that all testers gave for one particular soda flavor, and each column holds the responses of one person for all sodas.

The `SodaSurvey` program computes and prints the average responses for each soda and for each respondent. The sums of each soda and person are first stored in one-dimensional arrays of integers. Then the averages are computed and printed.

Multidimensional Arrays

An array can have one, two, three, or even more dimensions. Any array with more than one dimension is called a **multidimensional array** ⓘ.

It's fairly easy to picture a two-dimensional array as a table. A three-dimensional array could be drawn as a cube. However, once you are past three dimensions, multidimensional arrays might seem hard to visualize. Yet, consider that each subsequent dimension is simply a subdivision of the previous one. It is often best to think of larger multidimensional arrays in this way.

For example, suppose we wanted to store the number of students attending universities across the United States, broken down in a meaningful way. We might represent it as a four-dimensional array of integers. The first dimension represents the state. The second dimension represents the universities in each state. The third dimension represents the colleges in each university. Finally, the fourth dimension represents departments in each college. The value stored at each location is the number of students in one particular department. **Figure 8.5** ⓘ shows these subdivisions.

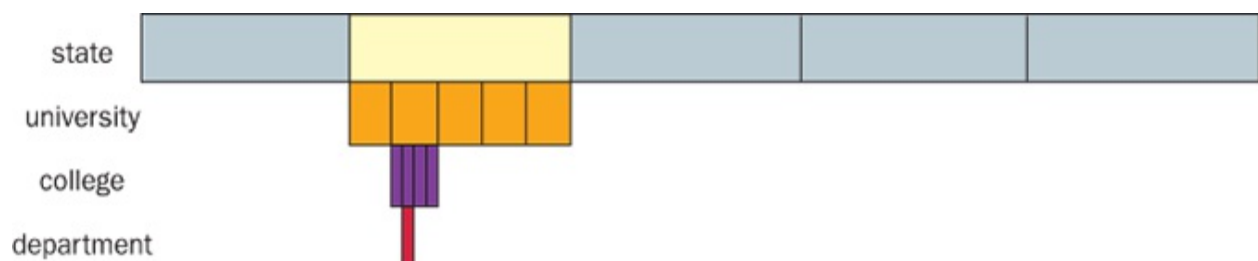


Figure 8.5 Visualization of a four-dimensional array

Key Concept

Using an array with more than two dimensions is rare in an object-oriented system.

Two-dimensional arrays are fairly common. However, care should be taken when deciding to create multidimensional arrays in a program. When dealing with large amounts of data that are managed at multiple levels, additional information and the methods needed to manage that information will probably be required. It is far more likely, for instance, that in the previous example, each state would be represented by an object, which may contain, among other things, an array to store information about each university, and so on.

There is one other important characteristic of Java arrays to consider. As we established previously, Java does not directly support multidimensional arrays. Instead, they are represented as arrays of references to array objects. Those arrays could themselves contain references to other arrays. This layering continues for as many dimensions as required. Because of this technique for representing each dimension, the arrays in any one dimension could be of different lengths. These are sometimes called *ragged arrays*. For example, the number of elements in each row of a two-dimensional array may not

be the same. In such situations, care must be taken to make sure the arrays are managed appropriately.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 8.23 How are multidimensional arrays implemented in Java?

SR 8.24 A two-dimensional array named `scores` holds the test scores for a class of students for a semester. Write code that prints out a single value that represents the range of scores held in the array. It prints out the value of the highest score minus the value of the lowest score. Each test score is represented as an integer.