

9 arrays

In our programming efforts, we often want to organize objects or primitive data in a form that is easy to access and modify. This chapter

introduces arrays, which are programming constructs that group data into lists. Arrays are a fundamental component of most high-level languages. We also explore the `ArrayList` class in the Java standard class library, which provides capabilities similar to arrays, with additional features.

chapter objectives

- ▶ Define and use arrays for basic data organization.
- ▶ Describe how arrays and array elements are passed as parameters.
- ▶ Explore how arrays and other objects can be combined to manage complex information.
- ▶ Describe the use of multidimensional arrays.
- ▶ Examine the `ArrayList` class and the costs of its versatility.

6.0 arrays

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.

array indexing

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 6.1 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.

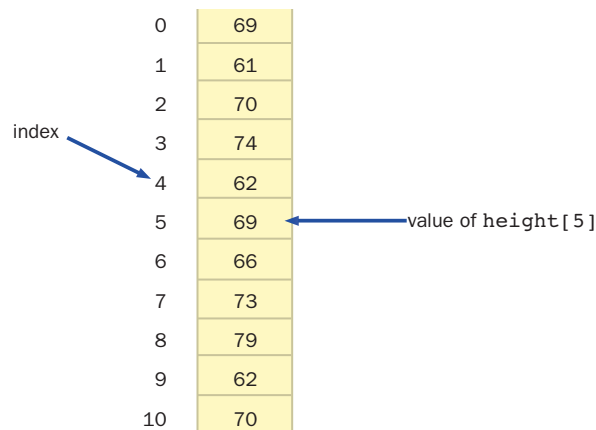
key
concept

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

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 Fig. 6.1 has 11 values, indexed from 0 to 10.

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]
```



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

figure 6.1 An array called `height` containing integer values

According to Fig. 6.1, `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)];
```

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 Fig. 6.1:

```
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.

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.

The example shown in Listing 6.1 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.

In Java, an array is an object. Memory space for the array elements is reserved by instantiating the array using the `new` operator.

key
concept

listing
6.1

```

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

public class BasicArray
{
    final static int LIMIT = 15;
    final static int MULTIPLE = 10;

    //-----
    // Creates an array, fills it with various integer values,
    // modifies one value, then prints them out.
    //-----
    public static void main (String[] args)
    {
        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

        for (int index = 0; index < LIMIT; index++)
            System.out.print (list[index] + " ");

        System.out.println ();
    }
}

```

output

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

Figure 6.2 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 con-



figure 6.2 The array list as it changes in the BasicArray program

stant is used to declare the size of the array, to control the `for` loop that initializes the array values, and to control the `for` loop that prints the values. The use of constants in this way is a good practice. It makes a program more readable and easier to modify. For instance, if the size of the array needed to change, only one line of code (the constant declaration) would need to be modified. We'll see another way to handle this situation in upcoming examples in this chapter.

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.

The index operator performs *automatic bounds checking*. Bounds checking 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

key
concept

Bounds checking ensures that an index used to refer to an array element is in range. The Java index operator performs automatic bounds checking.

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.

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. When referencing array elements, be careful to ensure that the index stays within the array bounds.

Another important characteristic of Java arrays is that their size is held in a constant called `length` in the array object. 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.

Let's look at another example. The program shown in Listing 6.2 reads 10 integers into an array called `numbers`, and then prints them in reverse order.

listing
6.2

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

import cs1.Keyboard;

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)
    {
        double[] numbers = new double[10];

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

listing
6.2 continued

```

    for (int index = 0; index < numbers.length; index++)
    {
        System.out.print ("Enter number " + (index+1) + ": ");
        numbers[index] = Keyboard.readDouble();
    }

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

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

    System.out.println ();
}
}

```

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 18.36

```

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 6.3, 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
6.3


```

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

import cs1.Keyboard;

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;

        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 = Keyboard.readString();

        // 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++)

```


Listing
6.3 continued

```
{
    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.

Both of 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.

We use the current character to calculate which index in the array to reference. Remember that each character has a numeric value based on the Unicode character set, and that 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. Consider the following declarations:

```
int total, sum, result;  
int[] grade1, grade2, grade3;
```

In the first declaration, the type of the three variables, `int`, is given at the beginning of the line. Similarly, in the second declaration, the type of the three variables, `int[]`, is also given at the beginning of the line. In both cases, the type applies to all variables in that particular declaration.

When the alternative form of array declaration is used, it can lead to potentially confusing situations, such as the following:

```
int grade1[], grade2, grade3[];
```

The variables `grade1` and `grade3` are declared to be arrays of integers, whereas `grade2` is a single integer. Although most declarations declare variables of the same type, this example declares variables of two different types. Why did the programmer write a declaration in this way? Is it a mistake? Should `grade2` be an array? This confusion is eliminated if the array brackets are associated with the element type. Therefore we associate the brackets with the element type throughout this text.

initializer lists

An important alternative technique for instantiating arrays is using an *initializer list* that provides 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.

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, 54, 65, 76, 87, 99};
```

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[] letterGrades = {'A', 'B', 'C', 'D', 'F'};
```

In this case, the variable `letterGrades` is declared to be an array of five characters, and the initializer list contains character literals. The program shown in Listing 6.4 demonstrates the use of an initializer list to instantiate an array.

An initializer list can be used to instantiate an array object instead of using the `new` operator. The size of the array and its initial values are determined by the initializer list.

key
concept



listing 6.4

```
//*****
// 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 scan = 0; scan < primeNums.length; scan++)
            System.out.print (primeNums[scan] + " ");

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

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 5.

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.

6.1 arrays of objects

In previous examples, the arrays stored 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.

arrays of string objects

Consider the following declaration:

```
String[] words = new String[25];
```

The variable `words` is an array of references to `String` objects. The `new` operator in the declaration instantiates the array and reserves space for 25 `String` references. Note that this declaration does not create any `String` objects; it merely creates an array that holds references to `String` objects.

The program called `GradeRange` shown in Listing 6.5 creates an array of `String` objects called `grades`, which stores letter grades for a course. The `String` objects are created using string literals in the initializer list. Note that this array could not have been declared as an array of characters because the plus and minus grades create two-character strings. The output for the `GradeRange` program shown in Listing 6.5 lists various letter grades and their corresponding lower numeric cutoff values, which have been stored in a corresponding array of integers.

listing
6.5

```

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

public class GradeRange
{
    //-----
    //  Stores the possible grades and their numeric lowest value,
    //  then prints them out.
    //-----
    public static void main (String[] args)
    {
        String[] grades = {"A", "A-", "B+", "B", "B-", "C+", "C", "C-",
                           "D+", "D", "D-", "F"};

        int[] cutoff = {95, 90, 87, 83, 80, 77, 73, 70, 67, 63, 60, 0};

        for (int level = 0; level < cutoff.length; level++)
            System.out.println (grades[level] + "\t" + cutoff[level]);
    }
}

```

output

```

A      95
A-     90
B+     87
B      83
B-     80
C+     77
C      73
C-     70
D+     67
D      63
D-     60
F      0

```

Sometimes two arrays with corresponding elements are called *parallel arrays*. The danger of parallel arrays is that one may become out of synch with the other. In an object-oriented approach, we would generally be better off creating one

array that held a single object containing all necessary information. For example, the `GradeRange` program could be changed to use a single array of objects that contain both the grade string and the numeric cutoff value. This modification is left as a programming project.

command-line arguments

The formal 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.

The Java runtime 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.

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

key
concept

The program shown in Listing 6.6 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.

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 would 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.

filling arrays of objects

We must always take into account an important characteristic of object arrays: The creation of the array and the creation of the objects that we store in the array are two separate steps. When we declare an array of `String` objects, for example,

listing
6.6


```

//*****
//  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]);
        System.out.println ();
    }
}

```

output

```

>java NameTag Howdy John

    Howdy
My name is John

>java NameTag Hello William

    Hello
My name is William

```

**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.

we create an array that holds `String` references. The `String` objects themselves must be created separately. In previous examples, the `String` objects were created using string literals in an initializer list, or, in the case of command-line arguments, they were created by the Java runtime environment.

This issue is demonstrated in the `Tunes` program and its accompanying classes. Listing 6.7 shows the `Tunes` class, which contains a `main` method that creates, modifies, and examines a compact disc (CD) collection. Each CD added to the collection is specified by its title, artist, purchase price, and number of tracks.

listing
6.7

```

//*****
//  Tunes.java      Author: Lewis/Loftus
//
//  Driver for demonstrating the use of an array of objects.
//*****

public class Tunes
{
    //-----
    //  Creates a CDCollection object and adds some CDs to it. Prints
    //  reports on the status of the collection.
    //-----
    public static void main (String[] args)
    {
        CDCollection music = new CDCollection ();

        music.addCD ("Storm Front", "Billy Joel", 14.95, 10);
        music.addCD ("Come On Over", "Shania Twain", 14.95, 16);
        music.addCD ("Soundtrack", "Les Miserables", 17.95, 33);
        music.addCD ("Graceland", "Paul Simon", 13.90, 11);

        System.out.println (music);

        music.addCD ("Double Live", "Garth Brooks", 19.99, 26);
        music.addCD ("Greatest Hits", "Jimmy Buffet", 15.95, 13);

        System.out.println (music);
    }
}

```

output

```

*****
My CD Collection

Number of CDs: 4
Total value: $61.75
Average cost: $15.44

CD List:

$14.95  10      Storm Front    Billy Joel
$14.95  16      Come On Over   Shania Twain
$17.95  33      Soundtrack     Les Miserables
$13.90  11      Graceland     Paul Simon

```

listing
6.7

continued

```

*****
My CD Collection

Number of CDs: 6
Total value: $97.69
Average cost: $16.28

CD List:

$14.95  10      Storm Front      Billy Joel
$14.95  16      Come On Over    Shania Twain
$17.95  33      Soundtrack      Les Miserables
$13.90  11      Graceland       Paul Simon
$19.99  26      Double Live     Garth Brooks
$15.95  13      Greatest Hits   Jimmy Buffet

```

Listing 6.8 shows the `CDCollection` class. It contains an array of `CD` objects representing the collection. It maintains a count of the CDs 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 CDs are added to the collection.

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

Each time a `CD` 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 `CD` 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 `CD` 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 `CD` collection. The user of the `CDCollection` object (the main method) never has to worry about running out of space because it's all handled internally.

Figure 6.3 shows a UML class diagram of the `Tunes` program. Recall that the open diamond indicates aggregation (a has-a relationship). The cardinality of the relationship is also noted: a `CDCollection` object contains zero or more `CD` objects.

listing
6.8


```

//*****
//  CDCollection.java      Author: Lewis/Loftus
//
//  Represents a collection of compact discs.
//*****

import java.text.NumberFormat;

public class CDCollection
{
    private CD[] collection;
    private int count;
    private double totalCost;

    //-----
    //  Creates an initially empty collection.
    //-----
    public CDCollection ()
    {
        collection = new CD[100];
        count = 0;
        totalCost = 0.0;
    }

    //-----
    //  Adds a CD to the collection, increasing the size of the
    //  collection if necessary.
    //-----
    public void addCD (String title, String artist, double cost,
                      int tracks)
    {
        if (count == collection.length)
            increaseSize();

        collection[count] = new CD (title, artist, cost, tracks);
        totalCost += cost;
        count++;
    }
}

```

listing
6.8 continued

```

//-----
// Returns a report describing the CD collection.
//-----
public String toString()
{
    NumberFormat fmt = NumberFormat.getCurrencyInstance();

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

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

    report += "\n\nCD List:\n\n";

    for (int cd = 0; cd < count; cd++)
        report += collection[cd].toString() + "\n";

    return report;
}

//-----
// Doubles the size of the collection by creating a larger array
// and copying the existing collection into it.
//-----
private void increaseSize ()
{
    CD[] temp = new CD[collection.length * 2];

    for (int cd = 0; cd < collection.length; cd++)
        temp[cd] = collection[cd];

    collection = temp;
}
}

```

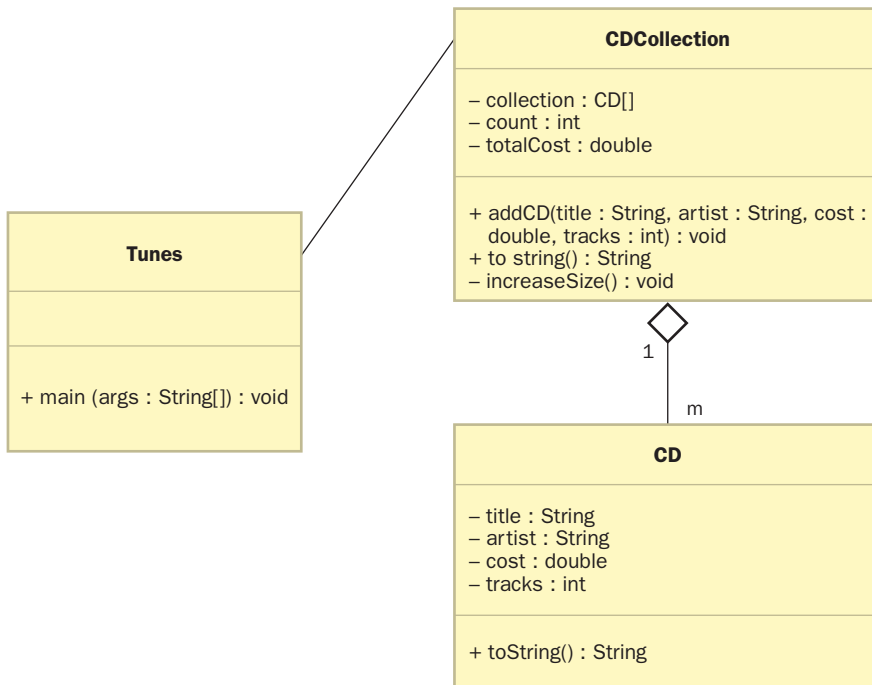


figure 6.3 A UML class diagram of the Tunes program

The `toString` method of the `CDCollection` class returns an entire report summarizing the collection. The report is created, in part, using calls to the `toString` method of each `CD` object stored in the collection. Listing 6.9 shows the `CD` class.

6.2 sorting

Sorting is the process of arranging a list of items in a well-defined order. For example, you may want to alphabetize a list of names or put a list of survey results into descending numeric order. Many sorting algorithms have been developed and critiqued over the years. In fact, sorting is considered to be a classic area of study in computer science.

listing
6.9


```

//*****
//  CD.java      Author: Lewis/Loftus
//
//  Represents a compact disc.
//*****

import java.text.NumberFormat;

public class CD
{
    private String title, artist;
    private double cost;
    private int tracks;

    //-----
    //  Creates a new CD with the specified information.
    //-----
    public CD (String name, String singer, double price, int numTracks)
    {
        title = name;
        artist = singer;
        cost = price;
        tracks = numTracks;
    }

    //-----
    //  Returns a description of this CD.
    //-----
    public String toString()
    {
        NumberFormat fmt = NumberFormat.getCurrencyInstance();

        String description;

        description = fmt.format(cost) + "\t" + tracks + "\t";
        description += title + "\t" + artist;

        return description;
    }
}

```

This section examines two sorting algorithms: selection sort and insertion sort. Complete coverage of various sorting techniques is beyond the scope of this text. Instead we introduce the topic and establish some of the fundamental ideas involved. We do not delve into a detailed analysis of the algorithms but instead focus on the strategies involved and general characteristics.

Selection sort and insertion sort are two sorting algorithms that define the processing steps for putting a list of values into a well-defined order.

key
concept

selection sort

The *selection sort* algorithm sorts a list of values by successively putting particular values in their final, sorted positions. In other words, for each position in the list, the algorithm selects the value that should go in that position and puts it there. Let's consider the problem of putting a list of numeric values into ascending order.

Selection sort works by putting each value in its final position, one at a time.

key
concept

The general strategy of selection sort is: Scan the entire list to find the smallest value. Exchange that value with the value in the first position of the list. Scan the rest of the list (all but the first value) to find the smallest value, then exchange it with the value in the second position of the list. Scan the rest of the list (all but the first two values) to find the smallest value, then exchange it with the value in the third position of the list. Continue this process for all but the last position in the list (which will end up containing the largest value). When the process is complete, the list is sorted. Figure 6.4 demonstrates the use of the selection sort algorithm.

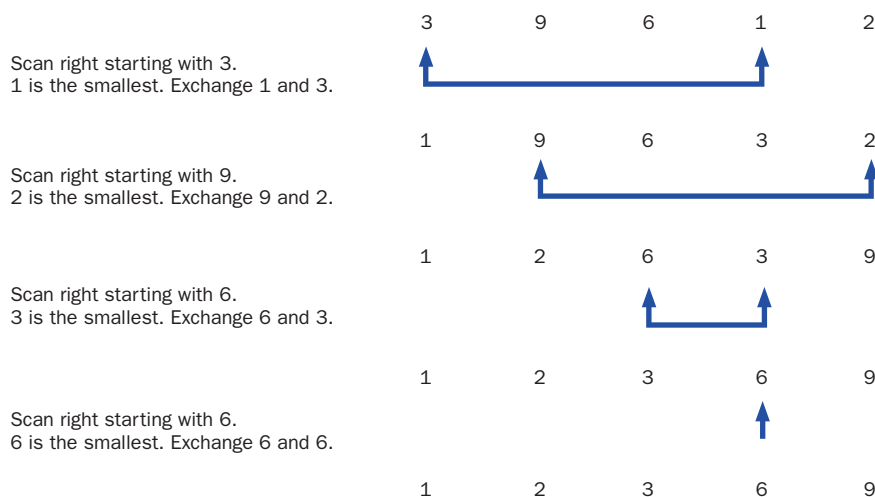


figure 6.4 Selection sort processing

The program shown in Listing 6.10 uses a selection sort to arrange a list of values into ascending order. The `SortGrades` class contains a `main` method that creates an array of integers. It calls the static method `selectionSort` in the `Sorts` class to put them in ascending order.

Listing 6.11 shows the `Sorts` class. It contains three static sorting algorithms. The `SortGrades` program uses only the `selectionSort` method. The other methods are discussed later in this section.

The implementation of the `selectionSort` method uses two `for` loops to sort an array of integers. The outer loop controls the position in the array where the next smallest value will be stored. The inner loop finds the smallest value in the rest of the list by scanning all positions greater than or equal to the index specified by the outer loop. When the smallest value is determined, it is exchanged with the value stored at the index. This exchange is done in three assignment statements by using an extra variable called `temp`. This type of exchange is often called *swapping*.

key
concept

Swapping is the process of exchanging two values. Swapping requires three assignment statements.

listing
6.10



```
//*****
//  SortGrades.java          Author: Lewis/Loftus
//
//  Driver for testing a numeric selection sort.
//*****

public class SortGrades
{
    //-----
    //  Creates an array of grades, sorts them, then prints them.
    //-----
    public static void main (String[] args)
    {
        int[] grades = {89, 94, 69, 80, 97, 85, 73, 91, 77, 85, 93};

        Sorts.selectionSort (grades);

        for (int index = 0; index < grades.length; index++)
            System.out.print (grades[index] + " ");
    }
}
```

output

```
69  73  77  80  85  85  89  91  93  94  97
```


listing
6.11

```

/*****
//  Sorts.java      Author: Lewis/Loftus
//
//  Demonstrates the selection sort and insertion sort algorithms,
//  as well as a generic object sort.
*****/

public class Sorts
{
    //-----
    //  Sorts the specified array of integers using the selection
    //  sort algorithm.
    //-----
    public static void selectionSort (int[] numbers)
    {
        int min, temp;

        for (int index = 0; index < numbers.length-1; index++)
        {
            min = index;
            for (int scan = index+1; scan < numbers.length; scan++)
                if (numbers[scan] < numbers[min])
                    min = scan;

            // Swap the values
            temp = numbers[min];
            numbers[min] = numbers[index];
            numbers[index] = temp;
        }
    }

    //-----
    //  Sorts the specified array of integers using the insertion
    //  sort algorithm.
    //-----
    public static void insertionSort (int[] numbers)
    {
        for (int index = 1; index < numbers.length; index++)
        {
            int key = numbers[index];
            int position = index;

```

listing
6.11 continued

```

        // shift larger values to the right
        while (position > 0 && numbers[position-1] > key)
        {
            numbers[position] = numbers[position-1];
            position--;
        }

        numbers[position] = key;
    }
}

//-----
// Sorts the specified array of objects using the insertion
// sort algorithm.
//-----
public static void insertionSort (Comparable[] objects)
{
    for (int index = 1; index < objects.length; index++)
    {
        Comparable key = objects[index];
        int position = index;

        // shift larger values to the right
        while (position > 0 && objects[position-1].compareTo(key) > 0)
        {
            objects[position] = objects[position-1];
            position--;
        }

        objects[position] = key;
    }
}
}

```

Note that because this algorithm finds the smallest value during each iteration, the result is an array sorted in ascending order (that is, smallest to largest). The algorithm can easily be changed to put values in descending order by finding the largest value each time.

insertion sort

The `Sorts` class also contains a method that performs an insertion sort on an array of integers. If used to sort the array of grades in the `SortGrades` program, it would produce the same results as the selection sort did. However, the processing to put the numbers in order is different.

The *insertion sort* algorithm sorts a list of values by repetitively inserting a particular value into a subset of the list that has already been sorted. One at a time, each unsorted element is inserted at the appropriate position in that sorted subset until the entire list is in order.

Insertion sort works by inserting each value into a previously sorted subset of the list.

key
concept

The general strategy of insertion sort is: Begin with a “sorted” list containing only one value. Sort the first two values in the list relative to each other by exchanging them if necessary. Insert the list’s third value into the appropriate position relative to the first two (sorted) values. Then insert the fourth value into its proper position relative to the first three values in the list. Each time an insertion is made, the number of values in the sorted subset increases by one. Continue this process until all values are inserted in their proper places, at which point the list is completely sorted.

The insertion process requires that the other values in the array shift to make room for the inserted element. Figure 6.5 demonstrates the behavior of the insertion sort algorithm.

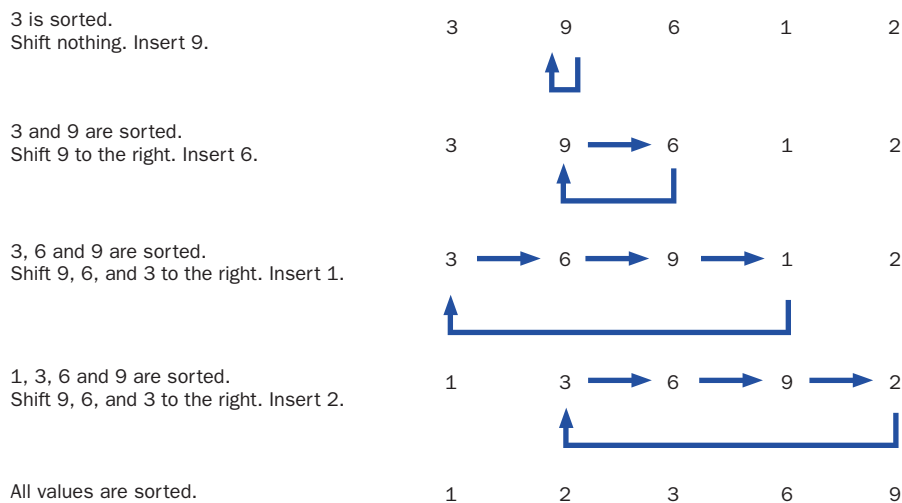


figure 6.5 Insertion sort processing

Similar to the selection sort implementation, the `insertionSort` method uses two `for` loops to sort an array of integers. In the insertion sort, however, the outer loop controls the index in the array of the next value to be inserted. The inner loop compares the current insert value with values stored at lower indexes (which make up a sorted subset of the entire list). If the current insert value is less than the value at `position`, that value is shifted to the right. Shifting continues until the proper position is opened to accept the insert value. Each iteration of the outer loop adds one more value to the sorted subset of the list, until the entire list is sorted.

sorting an array of objects

The `Sorts` class in Listing 6.11 contains an overloaded version of the `insertionSort` method. This version of the method accepts an array of `Comparable` objects and uses the insertion sort algorithm to put the objects in sorted order. Note the similarities in the general logic of both versions of the `insertionSort` method.

The main difference between the two versions of the `insertionSort` method is that one sorts an array of integers, whereas the other sorts an array of objects. We know what it means for one integer to be less than another integer, but what does it mean for one object to be less than another object? Basically, that decision depends on the objects being sorted and the characteristics on which the objects are to be ordered.

The key is that the parameter to the method is an array of `Comparable` objects. That is, the array is filled with objects that have implemented the `Comparable` interface, which we discussed in Chapter 5. Recall that the `Comparable` interface contains one method, `compareTo`, which is designed to return an integer that is less than zero, equal to zero, or greater than zero if the executing object is less than, equal to, or greater than the object to which it is being compared, respectively.

Let's look at an example. The `SortPhoneList` program shown in Listing 6.12 creates an array of `Contact` objects, sorts these objects using a call to the `insertionSort` method, and prints the sorted list.

Each `Contact` object represents a person with a last name, a first name, and a phone number. Listing 6.13 shows the `Contact` class.

listing
6.12


```

//*****
//  SortPhoneList.java      Author: Lewis/Loftus
//
//  Driver for testing an object sort.
//*****

public class SortPhoneList
{
    //-----
    //  Creates an array of Contact objects, sorts them, then prints
    //  them.
    //-----
    public static void main (String[] args)
    {
        Contact[] friends = new Contact[7];

        friends[0] = new Contact ("John", "Smith", "610-555-7384");
        friends[1] = new Contact ("Sarah", "Barnes", "215-555-3827");
        friends[2] = new Contact ("Mark", "Riley", "733-555-2969");
        friends[3] = new Contact ("Laura", "Getz", "663-555-3984");
        friends[4] = new Contact ("Larry", "Smith", "464-555-3489");
        friends[5] = new Contact ("Frank", "Phelps", "322-555-2284");
        friends[6] = new Contact ("Marsha", "Grant", "243-555-2837");

        Sorts.insertionSort(friends);

        for (int index = 0; index < friends.length; index++)
            System.out.println (friends[index]);
    }
}

```

output

Barnes, Sarah	215-555-3827
Getz, Laura	663-555-3984
Grant, Marsha	243-555-2837
Phelps, Frank	322-555-2284
Riley, Mark	733-555-2969
Smith, John	610-555-7384
Smith, Larry	464-555-3489

listing
6.13

```

//*****
//  Contact.java      Author: Lewis/Loftus
//
//  Represents a phone contact.
//*****

public class Contact implements Comparable
{
    private String firstName, lastName, phone;

    //-----
    //  Sets up this contact with the specified information.
    //-----
    public Contact (String first, String last, String telephone)
    {
        firstName = first;
        lastName = last;
        phone = telephone;
    }

    //-----
    //  Returns a description of this contact as a string.
    //-----
    public String toString ()
    {
        return lastName + ", " + firstName + "\t" + phone;
    }

    //-----
    //  Uses both last and first names to determine lexical ordering.
    //-----
    public int compareTo (Object other)
    {
        int result;

        if (lastName.equals(((Contact)other).lastName))
            result = firstName.compareTo(((Contact)other).firstName);
        else
            result = lastName.compareTo(((Contact)other).lastName);

        return result;
    }
}

```

The `Contact` class implements the `Comparable` interface and therefore provides a definition of the `compareTo` method. In this case, the contacts are sorted by last name; if two contacts have the same last name, their first names are used.

When the `insertionSort` method executes, it relies on the `compareTo` method of each object to determine the order. We are guaranteed that the objects in the array have implemented the `compareTo` method because they are all `Comparable` objects (according to the parameter type). The compiler will issue an error message if we attempt to pass an array to this method that does not contain `Comparable` objects. Therefore this version of the `insertionSort` method can be used to sort any array of objects as long as the objects have implemented the `Comparable` interface. This example demonstrates a classic and powerful use of interfaces to create generic algorithms that work on a variety of data.

comparing sorts

There are various reasons for choosing one sorting algorithm over another, including the algorithm's simplicity, its level of efficiency, the amount of memory it uses, and the type of data being sorted. An algorithm that is easier to understand is also easier to implement and debug. However, often the simplest sorts are the most inefficient ones. Efficiency is usually considered to be the primary criterion when comparing sorting algorithms. In general, one sorting algorithm is less efficient than another if it performs more comparisons than the other. There are several algorithms that are more efficient than the two we examined, but they are also more complex.

Sorting algorithms are ranked according to their efficiency, which is usually defined as the number of comparisons required to perform the sort.

key
concept

Both selection sort and insertion sort have essentially the same level of efficiency. Both have an outer loop and an inner loop with similar properties, if not purposes. The outer loop is executed once for each value in the list, and the inner loop compares the value in the outer loop with most, if not all, of the values in the rest of the list. Therefore, both algorithms perform approximately n^2 number of comparisons, where n is the number of values in the list. We say that both selection sort and insertion sort are algorithms of *order* n^2 . More efficient sorts perform fewer comparisons and are of a smaller order, such as $n \log_2 n$.

Both selection sort and insertion sort algorithms are of order n^2 . Other sorts are more efficient.

key
concept

Because both selection sort and insertion sort have the same general efficiency, the choice between them is almost arbitrary. However, there are some additional issues to consider. Selection sort is usually easy to understand and will often suffice in many situations. Further, each value moves exactly once to its final place

in the list. That is, although the selection and insertion sorts are equivalent (generally) in the number of comparisons made, selection sort makes fewer swaps.

**web
bonus**

The text's Web site contains a discussion and examples of additional sorting algorithms.

6.3 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 6.6 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.

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 6.14 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.

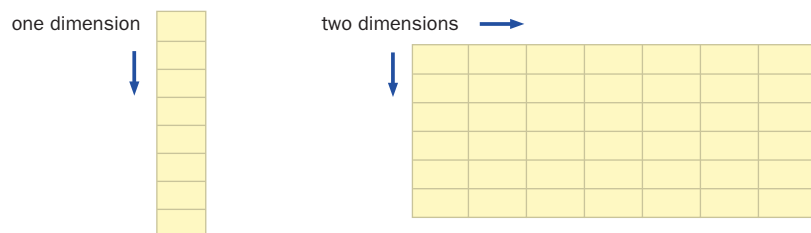


figure 6.6 A one-dimensional array and a two-dimensional array

listing
6.14


```

/*****
// 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
40	41	42	43	44	45	46	47	48	49

Nested `for` loops are used in the `TwoDArray` program to load the array with values and also to print those values in a table format. Carefully trace the processing to see how the nested loops eventually visit each element in the two-dimensional array. Note that the outer loops are governed by `table.length`, which represents the number of rows, and the inner loops are governed by `table[row].length`, which represents the number of columns in that row.

As with one-dimensional arrays, an initializer list can be used to instantiate a two-dimensional array, where each element is itself an array initializer list. This technique is used in the `SodaSurvey` program, which is shown in Listing 6.15.

listing 6.15



```

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

import java.text.DecimalFormat;

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];
    }
}

```

listing
6.15 continued

```
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 6.7 shows these subdivisions.

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.

key concept

Using an array with more than two dimensions is rare in an object-oriented system because intermediate levels are usually represented as separate objects.

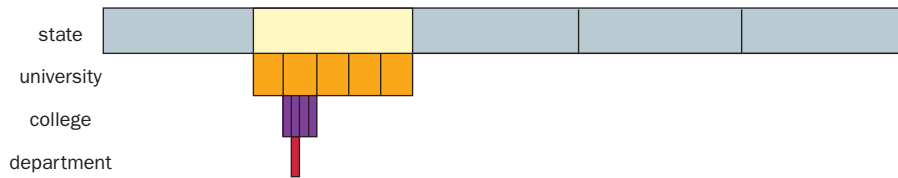


figure 6.7 Visualization of a four-dimensional array

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.

Each array in a given dimension of a multidimensional array could have a different length.

key
concept

6.4 the ArrayList class

The `ArrayList` class is part of the `java.util` package of the Java standard class library. It provides a service similar to an array in that it can store a list of values and reference them by an index. However, whereas an array remains a fixed size throughout its existence, an `ArrayList` object dynamically grows and shrinks as needed. A data element can be inserted into or removed from any location (index) of an `ArrayList` object with a single method invocation.

An `ArrayList` object is similar to an array, but it dynamically changes size as needed, and elements can be inserted and removed.

key
concept

The `ArrayList` class is part of the Collections API, a group of classes that serve to organize and manage other objects. We discuss collection classes further in Chapter 12.

Unlike an array, an `ArrayList` is not declared to store a particular type. An `ArrayList` object stores a list of references to the `Object` class. A reference to any type of object can be added to an `ArrayList` object. Because an `ArrayList` stores references, a primitive value must be stored in an appropriate wrapper class in order to be stored in an `ArrayList`. Figure 6.8 lists several methods of the `ArrayList` class.

```

ArrayList()
    Constructor: creates an initially empty list.

boolean add (Object obj)
    Inserts the specified object to the end of this list.

void add (int index, Object obj)
    Inserts the specified object into this list at the specified index.

void clear()
    Removes all elements from this list.

Object remove (int index)
    Removes the element at the specified index in this list and returns it.

Object get (int index)
    Returns the object at the specified index in this list without removing it.

int indexOf (Object obj)
    Returns the index of the first occurrence of the specified object.

boolean contains (Object obj)
    Returns true if this list contains the specified object.

boolean isEmpty()
    Returns true if this list contains no elements.

int size()
    Returns the number of elements in this list.

```

figure 6.8 Some methods of the ArrayList class

The program shown in Listing 6.16 instantiates an `ArrayList` called `band`. The method `add` is used to add several `String` objects to the `ArrayList` in a specific order. Then one particular string is deleted and another is inserted at a particular index. As with any other object, the `toString` method of the `ArrayList` class is automatically called whenever it is sent to the `println` method.

Note that when an element from an `ArrayList` is deleted, the list of elements “collapses” so that the indexes are kept continuous for the remaining elements. Likewise, when an element is inserted at a particular point, the indexes of the other elements are adjusted accordingly.

The objects stored in an `ArrayList` object can be of different reference types. The methods of the `ArrayList` class are designed to accept references to the `Object` class as parameters, thus allowing a reference to any kind of object to be passed to it. Note that an implication of this implementation is that the

listing
6.16

```

//*****
//  Beatles.java      Author: Lewis/Loftus
//
//  Demonstrates the use of a ArrayList object.
//*****

import java.util.ArrayList;

public class Beatles
{
    //-----
    //  Stores and modifies a list of band members.
    //-----
    public static void main (String[] args)
    {
        ArrayList band = new ArrayList();

        band.add ("Paul");
        band.add ("Pete");
        band.add ("John");
        band.add ("George");

        System.out.println (band);

        int location = band.indexOf ("Pete");
        band.remove (location);

        System.out.println (band);
        System.out.println ("At index 1: " + band.get(1));

        band.add (2, "Ringo");

        System.out.println (band);
        System.out.println ("Size of the band: " + band.size());
    }
}

```

output

```

[Paul, Pete, John, George]
[Paul, John, George]
At index 1: John
Size of the band: 4

```

`elementAt` method's return type is an `Object` reference. In order to retrieve a specific object from the `ArrayList`, the returned object must be cast to its original class. We discuss the `Object` class and its relationship to other classes in Chapter 7.

ArrayList efficiency

The `ArrayList` class is implemented, as you might imagine, using an array. That is, the `ArrayList` class stores as instance data an array of `Object` references. The methods provided by the class manipulate that array so that the indexes remain continuous as elements are added and removed.

When an `ArrayList` object is instantiated, the internal array is created with an initial capacity that defines the number of references it can currently handle. Elements can be added to the list without needing to allocate more memory until it reaches this capacity. When required, the capacity is expanded to accommodate the new need. We performed a similar operation in the Tunes program earlier in this chapter.

When an element is inserted into an `ArrayList`, all of the elements at higher indexes are copied into their new locations to make room for the new element. Figure 6.9 illustrates this process. Similar processing occurs when an element is removed from an `ArrayList`, except that the items are shifted in the other direction, closing the gap created by the deleted element to keep the indexes continuous. If several elements are inserted or deleted, this copying is repeated many times over.

If, in general, elements are added to or removed from the end of an `ArrayList`, its efficiency is not affected. But if elements are added to and/or removed from the front part of a long `ArrayList`, a huge amount of element copying will occur. An `ArrayList`, with its dynamic characteristics, is a useful abstraction of an array, but the abstraction masks some underlying activity that can be fairly inefficient depending on how it is used.

key concept

`ArrayList` processing can be inefficient depending on how it is used.

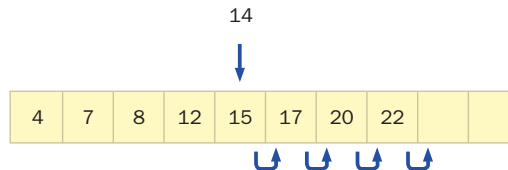


figure 6.9 Inserting an element into an `ArrayList` object

6.5 polygons and polylines

Arrays are helpful when drawing complex shapes. A polygon, for example, is a multisided shape that is defined in Java using a series of (x, y) points that indicate the vertices of the polygon. Arrays are often used to store the list of coordinates.

Polygons are drawn using methods of the `Graphics` class, similar to how we draw rectangles and ovals. Like these other shapes, a polygon can be drawn filled or unfilled. The methods used to draw a polygon are called `drawPolygon` and `fillPolygon`. Both of these methods are overloaded. One version uses arrays of integers to define the polygon, and the other uses an object of the `Polygon` class to define the polygon. We discuss the `Polygon` class later in this section.

In the version that uses arrays, the `drawPolygon` and `fillPolygon` methods take three parameters. The first is an array of integers representing the x coordinates of the points in the polygon, the second is an array of integers representing the corresponding y coordinates of those points, and the third is an integer that indicates how many points are used from each of the two arrays. Taken together, the first two parameters represent the (x, y) coordinates of the vertices of the polygons.

A polygon is always closed. A line segment is always drawn from the last point in the list to the first point in the list.

Similar to a polygon, a *polyline* contains a series of points connected by line segments. Polylines differ from polygons in that the first and last coordinates are not automatically connected when it is drawn. Since a polyline is not closed, it cannot be filled. Therefore there is only one method, called `drawPolyline`, used to draw a polyline.

As with the `drawPolygon` method, the first two parameters of the `drawPolyline` method are both arrays of integers. Taken together, the first two parameters represent the (x, y) coordinates of the end points of the line segments of the polyline. The third parameter is the number of points in the coordinate list.

The program shown in Listing 6.17 uses polygons to draw a rocket. The arrays called `xRocket` and `yRocket` define the points of the polygon that make up the main body of the rocket. The first point in the arrays is the upper tip of the rocket, and they progress clockwise from there. The `xWindow` and `yWindow` arrays specify the points for the polygon that form the window in the rocket. Both the rocket and the window are drawn as filled polygons.

A polygon is always a closed shape. The last point is automatically connected back to the first one.

key
concept

A polyline is similar to a polygon except that a polyline is not a closed shape.

key
concept



listing
6.17

```
/**
 * Rocket.java      Author: Lewis/Loftus
 *
 * Demonstrates the use of polygons and polylines.
 */

import javax.swing.JApplet;
import java.awt.*;

public class Rocket extends JApplet
{
    private final int APPLET_WIDTH = 200;
    private final int APPLET_HEIGHT = 200;

    private int[] xRocket = {100, 120, 120, 130, 130, 70, 70, 80, 80};
    private int[] yRocket = {15, 40, 115, 125, 150, 150, 125, 115, 40};

    private int[] xWindow = {95, 105, 110, 90};
    private int[] yWindow = {45, 45, 70, 70};

    private int[] xFlame = {70, 70, 75, 80, 90, 100, 110, 115, 120,
                           130, 130};
    private int[] yFlame = {155, 170, 165, 190, 170, 175, 160, 185,
                           160, 175, 155};

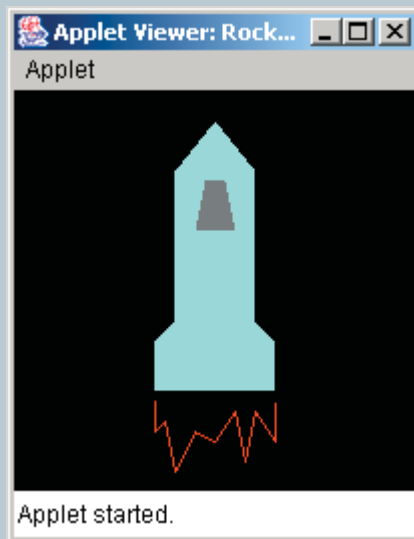
    //-----
    //  Sets up the basic applet environment.
    //-----
    public void init()
    {
        setBackground (Color.black);
        setSize (APPLET_WIDTH, APPLET_HEIGHT);
    }

    //-----
    //  Draws a rocket using polygons and polylines.
    //-----
    public void paint (Graphics page)
    {
        page.setColor (Color.cyan);
        page.fillPolygon (xRocket, yRocket, xRocket.length);

        page.setColor (Color.gray);
    }
}
```

listing
6.17 continued

```
page.fillPolygon (xWindow, yWindow, xWindow.length);  
  
page.setColor (Color.red);  
page.drawPolyline (xFlame, yFlame, xFlame.length);  
}  
}
```

display

The `xFlame` and `yFlame` arrays define the points of a polyline that are used to create the image of flame shooting out of the tail of the rocket. Because it is drawn as a polyline, and not a polygon, the flame is not closed or filled.

the Polygon class

A polygon can also be defined explicitly using an object of the `Polygon` class, which is defined in the `java.awt` package of the Java standard class library. Two versions of the overloaded `drawPolygon` and `fillPolygon` methods take a single `Polygon` object as a parameter.

A `Polygon` object encapsulates the coordinates of the polygon sides. The constructors of the `Polygon` class allow the creation of an initially empty polygon, or one defined by arrays of integers representing the point coordinates. The `Polygon` class contains methods to add points to the polygon and to determine whether a given point is contained within the polygon shape. It also contains methods to get a representation of a bounding rectangle for the polygon, as well as a method to translate all of the points in the polygon to another position. Figure 6.10 lists these methods.

```
Polygon ()  
    Constructor: Creates an empty polygon.  
  
Polygon (int[] xpoints, int[] ypoints, int npoints)  
  
    Constructor: Creates a polygon using the ( x, y) coordinate pairs  
    in corresponding entries of xpoints and ypoints.  
  
void addPoint (int x, int y)  
    Appends the specified point to this polygon.  
  
boolean contains (int x, int y)  
    Returns true if the specified point is contained in this polygon.  
  
boolean contains (Point p)  
    Returns true if the specified point is contained in this polygon.  
  
Rectangle getBounds ()  
    Gets the bounding rectangle for this polygon.  
  
void translate (int deltaX, int deltaY)  
    Translates the vertices of this polygon by deltaX along the x axis  
    and deltaY along the y axis.
```

figure 6.10 Some methods of the `Polygon` class

6.6 other button components

In the graphics track of Chapter 5, we introduced the basics of graphical user interface (GUI) construction: components, events, and listeners. Recall that the `JButton` class represents a push button. When pushed, an action event is generated and we can set up a listener to respond accordingly. Let's now examine some additional components—buttons of a different kind.

check boxes

A *check box* is a button that can be toggled on or off using the mouse, indicating that a particular boolean condition is set or unset. For example, a check box labeled `Collate` might be used to indicate whether the output of a print job should be collated. Although you might have a group of check boxes indicating a set of options, each check box operates independently. That is, each can be set to on or off and the status of one does not influence the others.

A check box allows the user to set the status of a boolean condition.

key
concept

The program in Listing 6.18 displays two check boxes and a label. The check boxes determine whether the text of the label is displayed in bold, italic, both, or neither. Any combination of bold and italic is valid. For example, both check boxes could be checked (on), in which case the text is displayed in both bold and italic. If neither is checked, the text of the label is displayed in a plain style.

The GUI for the `StyleOptions` program is embodied in the `StyleGUI` class shown in Listing 6.19. This organization is somewhat different than that used in the `Fahrenheit` program in the previous chapter. In this example, the frame is created in the `main` method. The `StyleGUI` object creates a panel on which the label and check boxes are arranged. The panel is returned to the `main` method using a call to `getPanel` and is added to the application frame.

A check box is represented by the `JCheckBox` class. When a check box changes state from selected (checked) to deselected (unchecked), or vice versa, it generates an *item event*. The `ItemListener` interface contains a single method called `itemStateChanged`. In this example, we use the same listener object to handle both check boxes.

listing
6.18

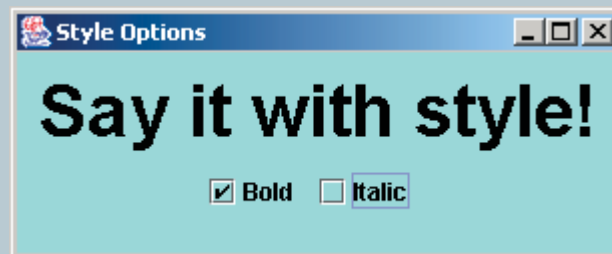
```
/**
 * StyleOptions.java      Author: Lewis/Loftus
 *
 * Demonstrates the use of check boxes.
 */

import javax.swing.*;

public class StyleOptions
{
    //-----
    //  Creates and presents the program frame.
    //-----
    public static void main (String[] args)
    {
        JFrame styleFrame = new JFrame ("Style Options");
        styleFrame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

        StyleGUI gui = new StyleGUI();
        styleFrame.getContentPane().add (gui.getPanel());

        styleFrame.pack();
        styleFrame.show();
    }
}
```

display

listing
6.19

```

//*****
//  StyleGUI.java      Author: Lewis/Loftus
//
//  Represents the user interface for the StyleOptions program.
//*****

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class StyleGUI
{
    private final int WIDTH = 300, HEIGHT = 100, FONT_SIZE = 36;
    private JLabel saying;
    private JCheckBox bold, italic;
    private JPanel primary;

    //-----
    //  Sets up a panel with a label and some check boxes that
    //  control the style of the label's font.
    //-----
    public StyleGUI()
    {
        saying = new JLabel ("Say it with style!");
        saying.setFont (new Font ("Helvetica", Font.PLAIN, FONT_SIZE));

        bold = new JCheckBox ("Bold");
        bold.setBackground (Color.cyan);
        italic = new JCheckBox ("Italic");
        italic.setBackground (Color.cyan);

        StyleListener listener = new StyleListener();
        bold.addItemListener (listener);
        italic.addItemListener (listener);

        primary = new JPanel();
        primary.add (saying);
        primary.add (bold);
        primary.add (italic);
        primary.setBackground (Color.cyan);
        primary.setPreferredSize (new Dimension(WIDTH, HEIGHT));
    }
}
```


listing
6.19 continued

```
//-----
// Returns the primary panel containing the GUI.
//-----
public JPanel getPanel()
{
    return primary;
}

//*****
// Represents the listener for both check boxes.
//*****
private class StyleListener implements ItemListener
{
    //-----
    // Updates the style of the label font style.
    //-----
    public void itemStateChanged (ItemEvent event)
    {
        int style = Font.PLAIN;

        if (bold.isSelected())
            style = Font.BOLD;

        if (italic.isSelected())
            style += Font.ITALIC;

        saying.setFont (new Font ("Helvetica", style, FONT_SIZE));
    }
}
}
```

This program also uses the `Font` class, which represents a particular *character font*. A `Font` object is defined by the font name, the font style, and the font size. The font name establishes the general visual characteristics of the characters. We are using the Helvetica font in this program. The style of a Java font can be plain, bold, italic, or bold and italic combined. The check boxes in our graphical user interface are set up to change the characteristics of our font style.

The style of a font is represented as an integer, and integer constants defined in the `Font` class are used to represent the various aspects of the style. The con-

stant `PLAIN` is used to represent a plain style. The constants `BOLD` and `ITALIC` are used to represent bold and italic, respectively. The sum of the `BOLD` and `ITALIC` constants indicates a style that is both bold and italic.

The `itemStateChanged` method of the listener determines what the revised style should be now that one of the check boxes has changed state. It initially sets the style to be plain. Then each check box is consulted in turn using the `isSelected` method, which returns a boolean value. First, if the bold check box is selected (checked), then the style is set to bold. Then, if the italic check box is selected, the `ITALIC` constant is added to the style variable. Finally, the font of the label is set to a new font with its revised style.

Note that, given the way the listener is written in this program, it doesn't matter which check box was clicked to generate the event. Both check boxes are processed by the same listener. It also doesn't matter whether the changed check box was toggled from selected to unselected or vice versa. The state of both check boxes is examined if either is changed.

radio buttons

A *radio button* is used with other radio buttons to provide a set of mutually exclusive options. Unlike a check box, a radio button is not useful by itself. It has meaning only when it is used with one or more other radio buttons. Only one option out of the group is valid. At any point in time, one and only one button of the group of radio buttons is selected (on). When a radio button from the group is pushed, the other button in the group that is currently on is automatically toggled off.

The term radio buttons comes from the way the buttons worked on an old-fashioned car radio. At any point, one button was pushed to specify the current choice of station; when another was pushed, the current one automatically popped out.

The `QuoteOptions` program, shown in Listing 6.20, displays a label and a group of radio buttons. The radio buttons determine which quote is displayed in the label. Because only one of the quotes can be displayed at a time, the use of radio buttons is appropriate. For example, if the `Comedy` radio button is selected, the comedy quote is displayed in the label. If the `Philosophy` button is then pressed, the `Comedy` radio button is automatically toggled off and the comedy quote is replaced by a philosophical one.

Radio buttons operate as a group, providing a set of mutually exclusive options. When one button is selected, the currently selected button is toggled off.

key
concept



listing
6.20

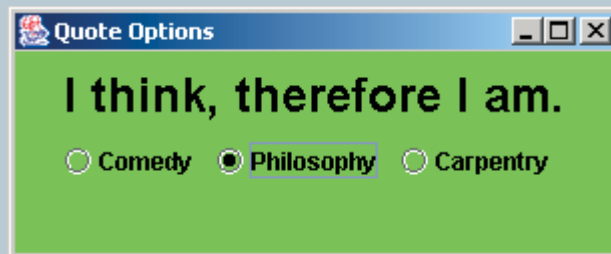
```
/**
 * QuoteOptions.java      Author: Lewis/Loftus
 *
 * Demonstrates the use of radio buttons.
 */

import javax.swing.*;

public class QuoteOptions
{
    //-----
    //  Creates and presents the program frame.
    //-----
    public static void main (String[] args)
    {
        JFrame quoteFrame = new JFrame ("Quote Options");
        quoteFrame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

        QuoteGUI gui = new QuoteGUI();
        quoteFrame.getContentPane().add (gui.getPanel());

        quoteFrame.pack();
        quoteFrame.show();
    }
}
```

display

The structure of this program is similar to that of the `StyleOptions` program from the previous section. The label and radio buttons are displayed on a panel defined in the `QuoteGUI` class, shown in Listing 6.21. A radio button is represented by the `JRadioButton` class. Because the radio buttons in a set work together, the `ButtonGroup` class is used to define a set of related radio buttons.

listing
6.21

```

//*****
//  QuoteGUI.java          Author: Lewis/Loftus
//
//  Represents the user interface for the QuoteOptions program.
//*****

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class QuoteGUI
{
    private final int WIDTH = 300, HEIGHT = 100;
    private JPanel primary;
    private JLabel quote;
    private JRadioButton comedy, philosophy, carpentry;
    private String comedyQuote = "Take my wife, please.";
    private String philosophyQuote = "I think, therefore I am.";
    private String carpentryQuote = "Measure twice. Cut once.";

    //-----
    //  Sets up a panel with a label and a set of radio buttons
    //  that control its text.
    //-----
    public QuoteGUI()
    {
        quote = new JLabel (comedyQuote);
        quote.setFont (new Font ("Helvetica", Font.BOLD, 24));

        comedy = new JRadioButton ("Comedy", true);
        comedy.setBackground (Color.green);
        philosophy = new JRadioButton ("Philosophy");
        philosophy.setBackground (Color.green);
        carpentry = new JRadioButton ("Carpentry");
        carpentry.setBackground (Color.green);
    }
}
```

listing
6.21 continued

```

    ButtonGroup group = new ButtonGroup();
    group.add (comedy);
    group.add (philosophy);
    group.add (carpentry);

    QuoteListener listener = new QuoteListener();
    comedy.addActionListener (listener);
    philosophy.addActionListener (listener);
    carpentry.addActionListener (listener);

    primary = new JPanel();
    primary.add (quote);
    primary.add (comedy);
    primary.add (philosophy);
    primary.add (carpentry);
    primary.setBackground (Color.green);
    primary.setPreferredSize (new Dimension(WIDTH, HEIGHT));
}

//-----
// Returns the primary panel containing the GUI.
//-----
public JPanel getPanel()
{
    return primary;
}

//*****
// Represents the listener for all radio buttons
//*****
private class QuoteListener implements ActionListener
{
    //-----
    // Sets the text of the label depending on which radio
    // button was pressed.
    //-----
    public void actionPerformed (ActionEvent event)
    {
        Object source = event.getSource();

```

Listing 6.21 continued

```
    if (source == comedy)
        quote.setText (comedyQuote);
    else
        if (source == philosophy)
            quote.setText (philosophyQuote);
        else
            quote.setText (carpentryQuote);
    }
}
```

Note that each button is added to the button group, and also that each button is added individually to the panel. A `ButtonGroup` object is not a container to organize and display components; it is simply a way to define the group of radio buttons that work together to form a set of dependent options. The `ButtonGroup` object ensures that the currently selected radio button is turned off when another in the group is selected.

A radio button produces an action event when it is selected. The `actionPerformed` method of the listener first determines the source of the event using the `getSource` method, and then compares it to each of the three radio buttons in turn. Depending on which button was selected, the text of the label is set to the appropriate quote.

Note that unlike push buttons, both check boxes and radio buttons are *toggle buttons*, meaning that at any time they are either on or off. The difference is in how they are used. Independent options (choose any combination) are controlled with check boxes. Dependent options (choose one of a set) are controlled with radio buttons. If there is only one option to be managed, a check box can be used by itself. As we mentioned earlier, a radio button, on the other hand, makes sense only in conjunction with one or more other radio buttons.

Also note that check boxes and radio buttons produce different types of events. A check box produces an item event and a radio button produces an action event. The use of different event types is related to the differences in button functionality. A check box produces an event when it is selected or deselected, and the listener could make the distinction if desired. A radio button, on the other hand, only produces an event when it is selected (the currently selected button from the group is deselected automatically).



summary of key concepts

- ▶ An array of size N is indexed from 0 to $N-1$.
- ▶ In Java, an array is an object. Memory space for the array elements is reserved by instantiating the array using the `new` operator.
- ▶ Bounds checking ensures that an index used to refer to an array element is in range. The Java index operator performs automatic bounds checking.
- ▶ An initializer list can be used to instantiate an array object instead of using the `new` operator. The size of the array and its initial values are determined by the initializer list.
- ▶ An entire array can be passed as a parameter, making the formal parameter an alias of the original.
- ▶ Command-line arguments are stored in an array of `String` objects and are passed to the `main` method.
- ▶ Instantiating an array of objects reserves room to store references only. The objects that are stored in each element must be instantiated separately.
- ▶ Selection sort and insertion sort are two sorting algorithms that define the processing steps for putting a list of values into a well-defined order.
- ▶ Selection sort works by putting each value in its final position, one at a time.
- ▶ Swapping is the process of exchanging two values. Swapping requires three assignment statements.
- ▶ Insertion sort works by inserting each value into a previously sorted subset of the list.
- ▶ Sorting algorithms are ranked according to their efficiency, which is usually defined as the number of comparisons required to perform the sort.
- ▶ Both selection sort and insertion sort algorithms are of order n^2 . Other sorts are more efficient.
- ▶ Using an array with more than two dimensions is rare in an object-oriented system because intermediate levels are usually represented as separate objects.
- ▶ Each array in a given dimension of a multidimensional array could have a different length.

- An `ArrayList` object is similar to an array, but it dynamically changes size as needed, and elements can be inserted and removed.
- `ArrayList` processing can be inefficient depending on how it is used.
- A polygon is always a closed shape. The last point is automatically connected back to the first one.
- A polyline is similar to a polygon except that a polyline is not a closed shape.
- A check box allows the user to set the status of a boolean condition.
- Radio buttons operate as a group, providing a set of mutually exclusive options. When one button is selected, the currently selected button is toggled off.

self-review questions

- 6.1 Explain the concept of array bounds checking. What happens when a Java array is indexed with an invalid value?
- 6.2 Describe the process of creating an array. When is memory allocated for the array?
- 6.3 What is an off-by-one error? How does it relate to arrays?
- 6.4 What does an array initializer list accomplish?
- 6.5 Can an entire array be passed as a parameter? How is this accomplished?
- 6.6 How is an array of objects created?
- 6.7 What is a command-line argument?
- 6.8 What are parallel arrays?
- 6.9 Which is better: selection sort or insertion sort? Explain.
- 6.10 How are multidimensional arrays implemented in Java?
- 6.11 What are the advantages of using an `ArrayList` object as opposed to an array? What are the disadvantages?
- 6.12 What is a polyline? How do we specify its shape?
- 6.13 Compare and contrast check boxes and radio buttons.
- 6.14 How does the `Timer` class help us perform animations in Java?

exercises

- 6.1 Which of the following are valid declarations? Which instantiate an array object? Explain your answers.

```
int primes = {2, 3, 4, 5, 7, 11};
float elapsedTimes[] = {11.47, 12.04, 11.72, 13.88};
int[] scores = int[30];
int[] primes = new {2,3,5,7,11};
int[] scores = new int[30];
char grades[] = {'a', 'b', 'c', 'd', 'f'};
char[] grades = new char[];
```

- 6.2 Describe five programs that are difficult to implement without using arrays.
- 6.3 Describe what problem occurs in the following code. What modifications should be made to it to eliminate the problem?

```
int[] numbers = {3, 2, 3, 6, 9, 10, 12, 32, 3, 12, 6};
for (int count = 1; count <= numbers.length; count++)
    System.out.println (numbers[count]);
```

- 6.4 Write an array declaration and any necessary supporting classes to represent the following statements:
- ▶ students' names for a class of 25 students
 - ▶ students' test grades for a class of 40 students
 - ▶ credit-card transactions that contain a transaction number, a merchant name, and a charge
 - ▶ students' names for a class and homework grades for each student
 - ▶ for each employee of the L&L International Corporation: the employee number, hire date, and the amount of the last five raises
- 6.5 Write a method called `sumArray` that accepts an array of floating point values and returns the sum of the values stored in the array.
- 6.6 Write a method called `switchThem` that accepts two integer arrays as parameters and switches the contents of the arrays. Take into account that the arrays may be of different sizes.
- 6.7 Describe a program for which you would use the `ArrayList` class instead of arrays to implement choices. Describe a program for which you would use arrays instead of the `ArrayList` class. Explain your choices.

- 6.8 Explain what would happen if the radio buttons used in the `QuoteOptions` program were not organized into a `ButtonGroup` object. Modify the program to test your answer.

programming projects

- 6.1 Design and implement an application that reads an arbitrary number of integers that are in the range 0 to 50 inclusive and counts how many occurrences of each are entered. After all input has been processed, print all of the values (with the number of occurrences) that were entered one or more times.
- 6.2 Modify the program from Programming Project 6.1 so that it works for numbers in the range between -25 and 25.
- 6.3 Rewrite the `Sorts` class so that both sorting algorithms put the values in descending order. Create a driver class with a `main` method to exercise the modifications.
- 6.4 Design and implement an application that creates a histogram that allows you to visually inspect the frequency distribution of a set of values. The program should read in an arbitrary number of integers that are in the range 1 to 100 inclusive; then produce a chart similar to the one below that indicates how many input values fell in the range 1 to 10, 11 to 20, and so on. Print one asterisk for each value entered.



1	-	10		*****
11	-	20		**
21	-	30		*****
31	-	40		
41	-	50		***
51	-	60		*****
61	-	70		**
71	-	80		*****
81	-	90		*****
91	-	100		*****

- 6.5 The lines in the histogram in Programming Project 6.4 will be too long if a large number of values is entered. Modify the program so that it prints an asterisk for every five values in each category. Ignore

leftovers. For example, if a category had 17 values, print three asterisks in that row. If a category had 4 values, do not print any asterisks in that row.



- 6.6 Design and implement an application that computes and prints the mean and standard deviation of a list of integers x_1 through x_n . Assume that there will be no more than 50 input values. Compute both the mean and standard deviation as floating point values, using the following formulas.

$$\text{mean} = \frac{\sum_{i=1}^n x_i}{n}$$

$$\text{sd} = \sqrt{\frac{\sum_{i=1}^n (x_i - \text{mean})^2}{n - 1}}$$

- 6.7 The L&L Bank can handle up to 30 customers who have savings accounts. Design and implement a program that manages the accounts. Keep track of key information and allow each customer to make deposits and withdrawals. Produce appropriate error messages for invalid transactions. *Hint:* you may want to base your accounts on the `Account` class from Chapter 4. Also provide a method to add 3 percent interest to all accounts whenever the method is invoked.
- 6.8 Modify the `GradeRange` program from this chapter so that it eliminates the use of parallel arrays. Instead, design a new class called `Grade` that stores both the grade string and its cutoff value. Set both values using the `Grade` constructor and provide methods that return the values. In the `main` method of the revised `GradeRange` program, populate a single array with `Grade` objects, and then produce the same output as the original `GradeRange` program did.



- 6.9 The programming projects of Chapter 4 discussed a `Card` class that represents a standard playing card. Create a class called `DeckOfCards` that stores 52 objects of the `Card` class. Include methods to shuffle the deck, deal a card, and report the number of cards left in the deck. The `shuffle` method should assume a full deck.

Create a driver class with a `main` method that deals each card from a shuffled deck, printing each card as it is dealt.

- 6.10 Use the `Question` class from Chapter 5 to define a `Quiz` class. A quiz can be composed of up to 25 questions. Define the `add` method of the `Quiz` class to add a question to a quiz. Define the `giveQuiz` method of the `Quiz` class to present each question in turn to the user, accept an answer for each one, and keep track of the results. Define a class called `QuizTime` with a `main` method that populates a quiz, presents it, and prints the final results.
- 6.11 Modify your answer to Programming Project 6.10 so that the complexity level of the questions given in the quiz is taken into account. Overload the `giveQuiz` method so that it accepts two integer parameters that specify the minimum and maximum complexity levels for the quiz questions and only presents questions in that complexity range. Modify the `main` method to demonstrate this feature.
- 6.12 Modify the `Tunes` program so that it keeps the CDs sorted by title. Use the general object sort defined in the `Sorts` class from this chapter.
- 6.13 Modify the `Sorts` class to include an overloaded version of the `SelectionSort` method that performs a general object sort. Modify the `SortPhoneList` program to test the new sort.
- 6.14 Design and implement an applet that graphically displays the processing of a selection sort. Use bars of various heights to represent the values being sorted. Display the set of bars after each swap. Put a delay in the processing of the sort to give the human observer a chance to see how the order of the values changes.
- 6.15 Repeat Programming Project 6.14 using an insertion sort.
- 6.16 Design a class that represents a star with a specified radius and color. Use a filled polygon to draw the star. Design and implement an applet that draws 10 stars of random radius in random locations.
- 6.17 Design a class that represents the visual representation of a car. Use polylines and polygons to draw the car in any graphics context and at any location. Create a main driver to display the car.
- 6.18 Modify the solution to Programming Project 6.17 so that it uses the `Polygon` class to represent all polygons used in the drawing.
- 6.19 Modify the `Fahrenheit` program from Chapter 5 so that it uses a structure similar to the `StyleOptions` and `QuoteOptions` programs



from this chapter. Specifically, create the application frame in the `main` method and add the GUI panel to it.

- 6.20 Modify the `StyleOptions` program in this chapter to allow the user to specify the size of the font. Use a text field to obtain the size.
- 6.21 Modify the `QuoteOptions` program in this chapter so that it provides three additional quote options. Use an array to store all of the quote strings.
- 6.22 Design and implement an applet that draws 20 circles, with the radius and location of each circle determined at random. If a circle does not overlap any other circle, draw that circle in black. If a circle overlaps one or more other circles, draw it in cyan. Use an array to store a representation of each circle, then determine the color of each circle. Two circles overlap if the distance between their center points is less than the sum of their radii.
- 6.23 Design and implement an applet that draws a checkerboard with five red and eight black checkers on it in various locations. Store the checkerboard as a two-dimensional array.
- 6.24 Modify the applet from Programming Project 6.23 so that the program determines whether any black checkers can jump any red checkers. Under the checkerboard, print (using `drawString`) the row and column position of all black checkers that have possible jumps.

answers to self-review questions

- 6.1 Whenever a reference is made to a particular array element, the index operator (the brackets that enclose the subscript) ensures that the value of the index is greater than or equal to zero and less than the size of the array. If it is not within the valid range, an `ArrayIndexOutOfBoundsException` is thrown.
- 6.2 Arrays are objects. Therefore, as with all objects, to create an array we first create a reference to the array (its name). We then instantiate the array itself, which reserves memory space to store the array elements. The only difference between a regular object instantiation and an array instantiation is the bracket syntax.
- 6.3 An off-by-one error occurs when a program's logic exceeds the boundary of an array (or similar structure) by one. These errors include forgetting to process a boundary element as well as attempt-

ing to process a nonexistent element. Array processing is susceptible to off-by-one errors because their indexes begin at zero and run to one less than the size of the array.

- 6.4 An array initializer list is used in the declaration of an array to set up the initial values of its elements. An initializer list instantiates the array object, so the `new` operator is needed.
- 6.5 An entire array can be passed as a parameter. Specifically, because an array is an object, a reference to the array is passed to the method. Any changes made to the array elements will be reflected outside of the method.
- 6.6 An array of objects is really an array of object references. The array itself must be instantiated, and the objects that are stored in the array must be created separately.
- 6.7 A command-line argument is data that is included on the command line when the interpreter is invoked to execute the program. Command-line arguments are another way to provide input to a program. They are accessed using the array of strings that is passed into the `main` method as a parameter.
- 6.8 Parallel arrays are two or more arrays whose corresponding elements are related in some way. Because parallel arrays can easily get out of synch if not managed carefully, it is often better to create a single array of objects that encapsulate the related elements.
- 6.9 Selection sort and insertion sort are generally equivalent in efficiency, because they both take about n^2 number of comparisons to sort a list of n numbers. Selection sort, though, generally makes fewer swaps. Several sorting algorithms are more efficient than either of these.
- 6.10 A multidimensional array is implemented in Java as an array of array objects. The arrays that are elements of the outer array could also contain arrays as elements. This nesting process could continue for as many levels as needed.
- 6.11 An `ArrayList` keeps the indexes of its objects continuous as they are added and removed, and an `ArrayList` dynamically increases its capacity as needed. In addition, an `ArrayList` is implemented so that it stores references to the `Object` class, which allows any object to be stored in it. A disadvantage of the `ArrayList` class is that it

copies a significant amount of data in order to insert and delete elements, and this process is inefficient.

- 6.12 A polyline is defined by a series of points that represent its vertices. The `drawPolyline` method takes three parameters to specify its shape. The first is an array of integers that represent the x coordinates of the points. The second is an array of integers that represent the y coordinates of the points. The third parameter is a single integer that indicates the number of points to be used from the arrays.
- 6.13 Both check boxes and radio buttons show a toggled state: either on or off. However, radio buttons work as a group in which only one can be toggled on at any point in time. Check boxes, on the other hand, represent independent options. They can be used alone or in a set in which any combination of toggled states is valid.
- 6.14 The `Timer` class represents an object that generates an action event at regular intervals. The programmer sets the interval delay. An animation can be set up to change its display every time the timer goes off.