CHAPTER

7

SINGLE-DIMENSIONAL ARRAYS

Objectives

- To describe why arrays are necessary in programming (§7.1).
- To declare array reference variables and create arrays (§§7.2.1–7.2.2).
- To obtain array size using `arrayRefVar.length` and know default values in an array (§7.2.3).
- To access array elements using indexes (§7.2.4).
- To declare, create, and initialize an array using an array initializer (§7.2.5).
- To program common array operations (displaying arrays, summing all elements, finding the minimum and maximum elements, random shuffling, and shifting elements) (§7.2.6).
- To simplify programming using the for each loops (§7.2.7).
- To apply arrays in application development (`AnalyzeNumbers`, `DeckOfCards`) (§§7.3–7.4).
- To copy contents from one array to another (§7.5).
- To develop and invoke methods with array arguments and return values (§§7.6–7.8).
- To define a method with a variable-length argument list (§7.9).
- To search elements using the linear (§7.10.1) or binary (§7.10.2) search algorithm.
- To sort an array using the selection sort approach (§7.11).
- To use the methods in the `java.util.Arrays` class (§7.12).
- To pass arguments to the main method from the command line (§7.13).

7.1 Introduction

A single array variable can reference a large collection of data.

problem

Often you will have to store a large number of values during the execution of a program. Suppose, for instance, that you need to read 100 numbers, compute their average, and find out how many numbers are above the average. Your program first reads the numbers and computes their average, then compares each number with the average to determine whether it is above the average. In order to accomplish this task, the numbers must all be stored in variables. You have to declare 100 variables and repeatedly write almost identical code 100 times. Writing a program this way would be impractical. So, how do you solve this problem?

why array?

An efficient, organized approach is needed. Java and most other high-level languages provide a data structure, the *array*, which stores a fixed-size sequential collection of elements of the same type. In the present case, you can store all 100 numbers into an array and access them through a single array variable.

This chapter introduces single-dimensional arrays. The next chapter will introduce two-dimensional and multidimensional arrays.

7.2 Array Basics

Once an array is created, its size is fixed. An array reference variable is used to access the elements in an array using an index.

index

An array is used to store a collection of data, but often we find it more useful to think of an array as a collection of variables of the same type. Instead of declaring individual variables, such as `number0`, `number1`, . . . , and `number99`, you declare one array variable such as `numbers` and use `numbers[0]`, `numbers[1]`, . . . , and `numbers[99]` to represent individual variables. This section introduces how to declare array variables, create arrays, and process arrays using indexes.

7.2.1 Declaring Array Variables

element type

To use an array in a program, you must declare a variable to reference the array and specify the array's *element type*. Here is the syntax for declaring an array variable:

```
elementType[] arrayRefVar;
```

The `elementType` can be any data type, and all elements in the array will have the same data type. For example, the following code declares a variable `myList` that references an array of double elements.

```
double[] myList;
```

Note

You can also use `elementType arrayRefVar[]` to declare an array variable. This style comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers. The style `elementType[] arrayRefVar` is preferred.

preferred syntax

7.2.2 Creating Arrays

Unlike declarations for primitive data type variables, the declaration of an array variable does not allocate any space in memory for the array. It creates only a storage location for the reference to an array. If a variable does not contain a reference to an array, the value of the variable is `null`. You cannot assign elements to an array unless it has already been created. After an

null

array variable is declared, you can create an array by using the `new` operator and assign its reference to the variable with the following syntax:

new operator

```
arrayRefVar = new elementType[arraySize];
```

This statement does two things: (1) it creates an array using `new elementType[arraySize]`; (2) it assigns the reference of the newly created array to the variable `arrayRefVar`.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement as:

```
elementType[] arrayRefVar = new elementType[arraySize];
```

or

```
elementType arrayRefVar[] = new elementType[arraySize];
```

Here is an example of such a statement:

```
double[] myList = new double[10];
```

This statement declares an array variable, `myList`, creates an array of ten elements of `double` type, and assigns its reference to `myList`. To assign values to the elements, use the syntax:

```
arrayRefVar[index] = value;
```

For example, the following code initializes the array.

```
myList[0] = 5.6;
myList[1] = 4.5;
myList[2] = 3.3;
myList[3] = 13.2;
myList[4] = 4.0;
myList[5] = 34.33;
myList[6] = 34.0;
myList[7] = 45.45;
myList[8] = 99.993;
myList[9] = 11123;
```

This array is illustrated in Figure 7.1.

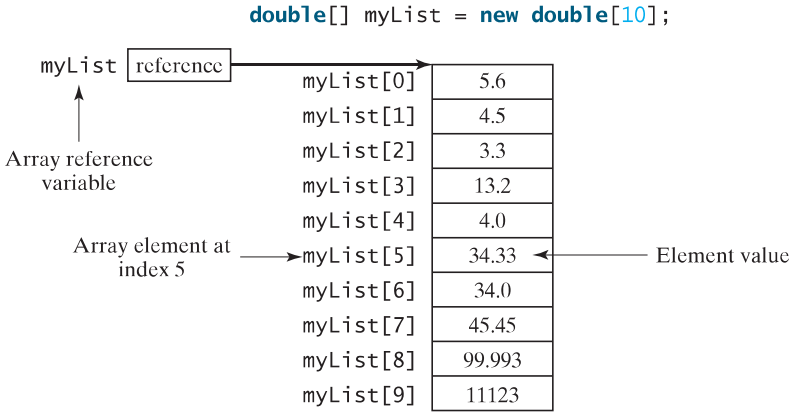


FIGURE 7.1 The array `myList` has ten elements of `double` type and `int` indices from `0` to `9`.

array vs. array variable

Note

An array variable that appears to hold an array actually contains a reference to that array. Strictly speaking, an array variable and an array are different, but most of the time the distinction can be ignored. Thus it is all right to say, for simplicity, that `myList` is an array, instead of stating, at greater length, that `myList` is a variable that contains a reference to an array of ten double elements.

7.2.3 Array Size and Default Values

array length

When space for an array is allocated, the array size must be given, specifying the number of elements that can be stored in it. The size of an array cannot be changed after the array is created. Size can be obtained using `arrayRefVar.length`. For example, `myList.length` is `10`.

default values

When an array is created, its elements are assigned the default value of `0` for the numeric primitive data types, `\u0000` for `char` types, and `false` for `boolean` types.

7.2.4 Accessing Array Elements

0 based

The array elements are accessed through the index. Array indices are `0` based; that is, they range from `0` to `arrayRefVar.length-1`. In the example in Figure 7.1, `myList` holds ten `double` values, and the indices are from `0` to `9`.

indexed variable

Each element in the array is represented using the following syntax, known as an *indexed variable*:

```
arrayRefVar[index];
```

For example, `myList[9]` represents the last element in the array `myList`.

Caution

Some programming languages use parentheses to reference an array element, as in `myList(9)`, but Java uses brackets, as in `myList[9]`.

An indexed variable can be used in the same way as a regular variable. For example, the following code adds the values in `myList[0]` and `myList[1]` to `myList[2]`.

```
myList[2] = myList[0] + myList[1];
```

The following loop assigns `0` to `myList[0]`, `1` to `myList[1]`, . . . , and `9` to `myList[9]`:

```
for (int i = 0; i < myList.length; i++) {
  myList[i] = i;
}
```

7.2.5 Array Initializers

array initializer

Java has a shorthand notation, known as the *array initializer*, which combines the declaration, creation, and initialization of an array in one statement using the following syntax:

```
elementType[] arrayRefVar = {value0, value1, ..., valuek};
```

For example, the statement

```
double[] myList = {1.9, 2.9, 3.4, 3.5};
```

declares, creates, and initializes the array `myList` with four elements, which is equivalent to the following statements:

```
double[] myList = new double[4];
myList[0] = 1.9;
myList[1] = 2.9;
```

```
myList[2] = 3.4;
myList[3] = 3.5;
```

Caution

The **new** operator is not used in the array-initializer syntax. Using an array initializer, you have to declare, create, and initialize the array all in one statement. Splitting it would cause a syntax error. Thus, the next statement is wrong:

```
double[] myList;
myList = {1.9, 2.9, 3.4, 3.5};
```

7.2.6 Processing Arrays

When processing array elements, you will often use a **for** loop—for two reasons:

- All of the elements in an array are of the same type. They are evenly processed in the same fashion repeatedly using a loop.
- Since the size of the array is known, it is natural to use a **for** loop.

Assume the array is created as follows:

```
double[] myList = new double[10];
```

The following are some examples of processing arrays.

1. *Initializing arrays with input values:* The following loop initializes the array **myList** with user input values.

```
java.util.Scanner input = new java.util.Scanner(System.in);
System.out.print("Enter " + myList.length + " values: ");
for (int i = 0; i < myList.length; i++)
  myList[i] = input.nextDouble();
```

2. *Initializing arrays with random values:* The following loop initializes the array **myList** with random values between **0.0** and **100.0**, but less than **100.0**.

```
for (int i = 0; i < myList.length; i++) {
  myList[i] = Math.random() * 100;
}
```

3. *Displaying arrays:* To print an array, you have to print each element in the array using a loop like the following:

```
for (int i = 0; i < myList.length; i++) {
  System.out.print(myList[i] + " ");
}
```

Tip

print character array

For an array of the **char[]** type, it can be printed using one print statement. For example, the following code displays **Dallas**:

```
char[] city = {'D', 'a', 'l', 'l', 'a', 's'};
System.out.println(city);
```

4. *Summing all elements:* Use a variable named **total** to store the sum. Initially **total** is **0**. Add each element in the array to **total** using a loop like this:

```
double total = 0;
for (int i = 0; i < myList.length; i++) {
  total += myList[i];
}
```

5. *Finding the largest element:* Use a variable named `max` to store the largest element. Initially `max` is `myList[0]`. To find the largest element in the array `myList`, compare each element with `max`, and update `max` if the element is greater than `max`.

```
double max = myList[0];
for (int i = 1; i < myList.length; i++) {
  if (myList[i] > max) max = myList[i];
}
```

6. *Finding the smallest index of the largest element:* Often you need to locate the largest element in an array. If an array has multiple elements with the same largest value, find the smallest index of such an element. Suppose the array `myList` is {`1`, `5`, `3`, `4`, `5`, `5`}. The largest element is `5` and the smallest index for `5` is `1`. Use a variable named `max` to store the largest element and a variable named `indexOfMax` to denote the index of the largest element. Initially `max` is `myList[0]`, and `indexOfMax` is `0`. Compare each element in `myList` with `max`, and update `max` and `indexOfMax` if the element is greater than `max`.

```
double max = myList[0];
int indexOfMax = 0;
for (int i = 1; i < myList.length; i++) {
  if (myList[i] > max) {
    max = myList[i];
    indexOfMax = i;
  }
}
```

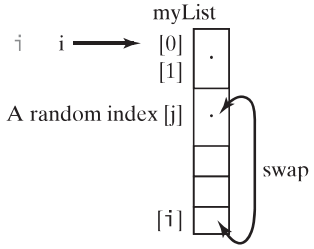
Random shuffling

VideoNote
Random shuffling

7. *Random shuffling:* In many applications, you need to randomly reorder the elements in an array. This is called *shuffling*. To accomplish this, for each element `myList[i]`, randomly generate an index `j` and swap `myList[i]` with `myList[j]`, as follows:

```
for (int i = myList.length - 1; i > 0; i--) {
  // Generate an index j randomly with 0 <= j <= i
  int j = (int)(Math.random()
    * (i + 1));

  // Swap myList[i] with myList[j]
  double temp = myList[i];
  myList[i] = myList[j];
  myList[j] = temp;
}
```

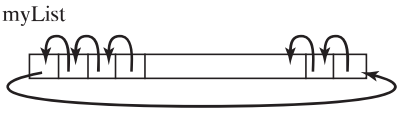


8. *Shifting elements:* Sometimes you need to shift the elements left or right. Here is an example of shifting the elements one position to the left and filling the last element with the first element:

```
double temp = myList[0]; // Retain the first element

// Shift elements left
for (int i = 1; i < myList.length; i++) {
  myList[i - 1] = myList[i];
}

// Move the first element to fill in the last position
myList[myList.length - 1] = temp;
```



9. *Simplifying coding:* Arrays can be used to greatly simplify coding for certain tasks. For example, suppose you wish to obtain the English name of a given month by its number. If the month names are stored in an array, the month name for a given month can be

accessed simply via the index. The following code prompts the user to enter a month number and displays its month name:

```
String[] months = {"January", "February", ..., "December"};
System.out.print("Enter a month number (1 to 12): ");
int monthNumber = input.nextInt();
System.out.println("The month is " + months[monthNumber - 1]);
```

If you didn't use the `months` array, you would have to determine the month name using a lengthy multi-way `if-else` statement as follows:

```
if (monthNumber == 1)
  System.out.println("The month is January");
else if (monthNumber == 2)
  System.out.println("The month is February");
...
else
  System.out.println("The month is December");
```

7.2.7 Foreach Loops

Java supports a convenient `for` loop, known as a *foreach loop*, which enables you to traverse the array sequentially without using an index variable. For example, the following code displays all the elements in the array `myList`:

```
for (double e: myList) {
  System.out.println(e);
}
```

You can read the code as "for each element `e` in `myList`, do the following." Note that the variable, `e`, must be declared as the same type as the elements in `myList`.

In general, the syntax for a foreach loop is

```
for (elementType element: arrayRefVar) {
  // Process the element
}
```

You still have to use an index variable if you wish to traverse the array in a different order or change the elements in the array.

Caution

Accessing an array out of bounds is a common programming error that throws a runtime `ArrayIndexOutOfBoundsException`. To avoid it, make sure that you do not use an index beyond `arrayRefVar.length - 1`.

ArrayIndexOutOfBounds-Exception

Programmers often mistakenly reference the first element in an array with index `1`, but it should be `0`. This is called the *off-by-one error*. Another common off-by-one error in a loop is using `<=` where `<` should be used. For example, the following loop is wrong.

off-by-one error

```
for (int i = 0; i <= list.length; i++)
  System.out.print(list[i] + " ");
```

The `<=` should be replaced by `<`.

7.1 How do you declare an array reference variable and how do you create an array?

7.2 When is the memory allocated for an array?

7.3 What is the output of the following code?

```
int x = 30;
int[] numbers = new int[x];
x = 60;
System.out.println("x is " + x);
System.out.println("The size of numbers is " + numbers.length);
```

7.4 Indicate `true` or `false` for the following statements:

- Every element in an array has the same type.
- The array size is fixed after an array reference variable is declared.
- The array size is fixed after it is created.
- The elements in an array must be a primitive data type.

7.5 Which of the following statements are valid?

```
int i = new int(30);
double d[] = new double[30];
char[] r = new char(1..30);
int i[] = (3, 4, 3, 2);
float f[] = {2.3, 4.5, 6.6};
char[] c = new char();
```

7.6 How do you access elements in an array?

7.7 What is the array index type? What is the lowest index? What is the representation of the third element in an array named `a`?

7.8 Write statements to do the following:

a. Create an array to hold `10` double values.

b. Assign the value `5.5` to the last element in the array.

c. Display the sum of the first two elements.

d. Write a loop that computes the sum of all elements in the array.

e. Write a loop that finds the minimum element in the array.

f. Randomly generate an index and display the element of this index in the array.

g. Use an array initializer to create another array with the initial values `3.5`, `5.5`, `4.52`, and `5.6`.

7.9 What happens when your program attempts to access an array element with an invalid index?

7.10 Identify and fix the errors in the following code:

```
1  public class Test {
2    public static void main(String[] args) {
3      double[100] r;
4
5      for (int i = 0; i < r.length(); i++);
6        r(i) = Math.random * 100;
7    }
8  }
```

7.11 What is the output of the following code?

```
1  public class Test {
2    public static void main(String[] args) {
3      int list[] = {1, 2, 3, 4, 5, 6};
```

```
4       for (int i = 1; i < list.length; i++)
5         list[i] = list[i - 1];
6
7       for (int i = 0; i < list.length; i++)
8         System.out.print(list[i] + " ");
9     }
10  }
```

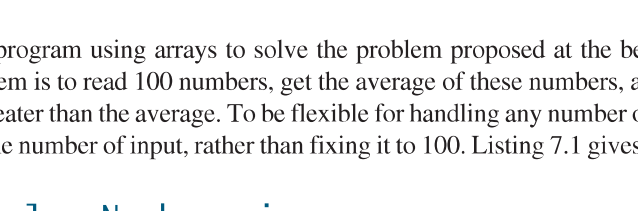
7.3 Case Study: Analyzing Numbers

The problem is to write a program that finds the number of items above the average of all items.

Now you can write a program using arrays to solve the problem proposed at the beginning of this chapter. The problem is to read 100 numbers, get the average of these numbers, and find the number of the items greater than the average. To be flexible for handling any number of input, we will let the user enter the number of input, rather than fixing it to 100. Listing 7.1 gives a solution.

LISTING 7.1 AnalyzeNumbers.java

```
1  public class AnalyzeNumbers {
2    public static void main(String[] args) {
3      java.util.Scanner input = new java.util.Scanner(System.in);
4      System.out.print("Enter the number of items: ");
5      int n = input.nextInt();
6      double [] numbers = new double[n];
7      double sum = 0;
8
9      System.out.print("Enter the numbers: ");
10     for (int i = 0; i < n; i++) {
11       numbers[i] = input.nextDouble();
12       sum += numbers[i];
13     }
14
15     double average = sum / n;
16
17     int count = 0; // The number of elements above average
18     for (int i = 0; i < n; i++)
19       if (numbers[i] > average)
20         count++;
21
22     System.out.println("Average is " + average);
23     System.out.println("Number of elements above the average is "
24       + count);
25   }
26 }
```



create array

store number in array

get average

above average?

```
Enter the number of items: 10 Enter
Enter the numbers: 3.4 5 6 1 6.5 7.8 3.5 8.5 6.3 9.5 Enter
Average is 5.75
Number of elements above the average is 6
```

The program prompts the user to enter the array size (line 5) and creates an array with the specified size (line 6). The program reads the input, stores numbers into the array (line 11), adds each number to `sum` in line 11, and obtains the average (line 15). It then compares

each number in the array with the average to count the number of values above the average (lines 17–20).

7.4 Case Study: Deck of Cards

The problem is to create a program that will randomly select four cards from a deck of cards.

Say you want to write a program that will pick four cards at random from a deck of `52` cards. All the cards can be represented using an array named `deck`, filled with initial values `0` to `51`, as follows:


```
int[] deck = new int[52];

// Initialize cards
for (int i = 0; i < deck.length; i++)
  deck[i] = i;
```

Card numbers `0` to `12`, `13` to `25`, `26` to `38`, and `39` to `51` represent 13 Spades, 13 Hearts, 13 Diamonds, and 13 Clubs, respectively, as shown in Figure 7.2. `cardNumber / 13` determines the suit of the card and `cardNumber % 13` determines the rank of the card, as shown in Figure 7.3. After shuffling the array `deck`, pick the first four cards from `deck`. The program displays the cards from these four card numbers.

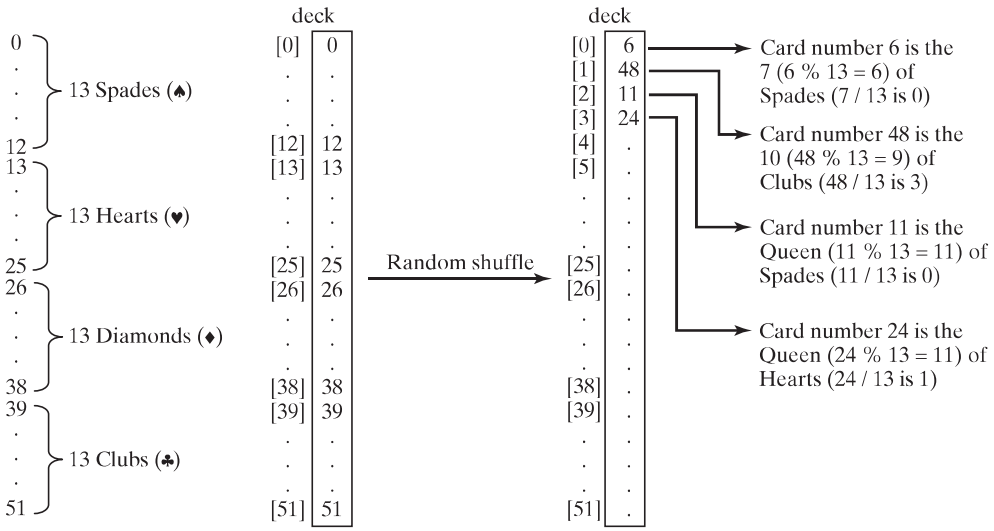


FIGURE 7.2 52 cards are stored in an array named `deck`.

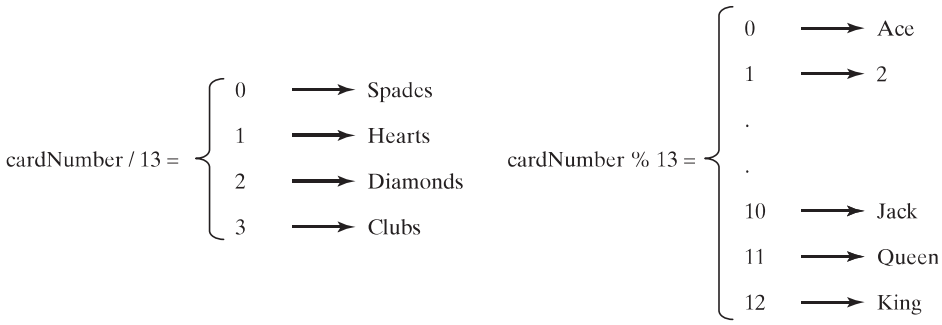


FIGURE 7.3 `CardNumber` identifies a card's suit and rank number.

Listing 7.2 gives the solution to the problem.

LISTING 7.2 DeckOfCards.java

```
1  public class DeckOfCards {
2    public static void main(String[] args) {
3      int[] deck = new int[52];                                          create array deck
4      String[] suits = {"Spades", "Hearts", "Diamonds", "Clubs"};        array of strings
5      String[] ranks = {"Ace", "2", "3", "4", "5", "6", "7", "8", "9",   array of strings
6        "10", "Jack", "Queen", "King"};
7
8      // Initialize the cards
9      for (int i = 0; i < deck.length; i++)                              initialize deck
10       deck[i] = i;
11
12     // Shuffle the cards
13     for (int i = 0; i < deck.length; i++) {                            shuffle deck
14       // Generate an index randomly
15       int index = (int)(Math.random() * deck.length);
16       int temp = deck[i];
17       deck[i] = deck[index];
18       deck[index] = temp;
19     }
20
21     // Display the first four cards
22     for (int i = 0; i < 4; i++) {
23       String suit = suits[deck[i] / 13];                               suit of a card
24       String rank = ranks[deck[i] % 13];                               rank of a card
25       System.out.println("Card number " + deck[i] + ": "
26         + rank + " of " + suit);
27     }
28   }
29 }
```

```
Card number 6: 7 of Spades
Card number 48: 10 of Clubs
Card number 11: Queen of Spades
Card number 24: Queen of Hearts
```

The program creates an array **suits** for four suits (line 4) and an array **ranks** for 13 cards in a suit (lines 5–6). Each element in these arrays is a string.

The program initializes **deck** with values **0** to **51** in lines 9–10. The **deck** value **0** represents the card Ace of Spades, **1** represents the card 2 of Spades, **13** represents the card Ace of Hearts, and **14** represents the card 2 of Hearts.

Lines 13–19 randomly shuffle the deck. After a deck is shuffled, **deck[i]** contains an arbitrary value. **deck[i] / 13** is **0**, **1**, **2**, or **3**, which determines the suit (line 23). **deck[i] % 13** is a value between **0** and **12**, which determines the rank (line 24). If the **suits** array is not defined, you would have to determine the suit using a lengthy multi-way **if-else** statement as follows:

```
if (deck[i] / 13 == 0)
  System.out.print("suit is Spades");
else if (deck[i] / 13 == 1)
  System.out.print("suit is Hearts");
else if (deck[i] / 13 == 2)
  System.out.print("suit is Diamonds");
else
  System.out.print("suit is Clubs");
```

With `suits = {"Spades", "Hearts", "Diamonds", "Clubs"}` created in an array, `suits[deck / 13]` gives the suit for the `deck`. Using arrays greatly simplifies the solution for this program.

7.12 Will the program pick four random cards if you replace lines 22–27 in Listing 7.2 DeckOfCards.java with the following code?

```
for (int i = 0; i < 4; i++) {
  int cardNumber = (int)(Math.random() * deck.length);
  String suit = suits[cardNumber / 13];
  String rank = ranks[cardNumber % 13];
  System.out.println("Card number " + cardNumber + ": "
    + rank + " of " + suit);
}
```

7.5 Copying Arrays

To copy the contents of one array into another, you have to copy the array's individual elements into the other array.

Often, in a program, you need to duplicate an array or a part of an array. In such cases you could attempt to use the assignment statement (`=`), as follows:

```
list2 = list1;
```

copy reference

However, this statement does not copy the contents of the array referenced by `list1` to `list2`, but instead merely copies the reference value from `list1` to `list2`. After this statement, `list1` and `list2` reference the same array, as shown in Figure 7.4. The array previously referenced by `list2` is no longer referenced; it becomes garbage, which will be automatically collected by the Java Virtual Machine (this process is called *garbage collection*).

garbage collection

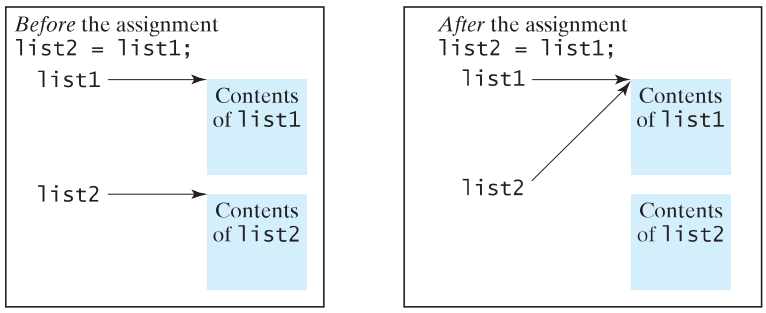


FIGURE 7.4 Before the assignment statement, `list1` and `list2` point to separate memory locations. After the assignment, the reference of the `list1` array is passed to `list2`.

In Java, you can use assignment statements to copy primitive data type variables, but not arrays. Assigning one array variable to another array variable actually copies one reference to another and makes both variables point to the same memory location.

There are three ways to copy arrays:

- Use a loop to copy individual elements one by one.
- Use the static `arraycopy` method in the `System` class.
- Use the `clone` method to copy arrays; this will be introduced in Chapter 13, Abstract Classes and Interfaces.

You can write a loop to copy every element from the source array to the corresponding element in the target array. The following code, for instance, copies `sourceArray` to `targetArray` using a `for` loop.

```
int[] sourceArray = {2, 3, 1, 5, 10};
int[] targetArray = new int[sourceArray.length];
for (int i = 0; i < sourceArray.length; i++) {
  targetArray[i] = sourceArray[i];
}
```

arraycopy method

Another approach is to use the `arraycopy` method in the `java.lang.System` class to copy arrays instead of using a loop. The syntax for `arraycopy` is:

```
arraycopy(sourceArray, srcPos, targetArray, tarPos, length);
```

The parameters `srcPos` and `tarPos` indicate the starting positions in `sourceArray` and `targetArray`, respectively. The number of elements copied from `sourceArray` to `targetArray` is indicated by `length`. For example, you can rewrite the loop using the following statement:

```
System.arraycopy(sourceArray, 0, targetArray, 0, sourceArray.length);
```

The `arraycopy` method does not allocate memory space for the target array. The target array must have already been created with its memory space allocated. After the copying takes place, `targetArray` and `sourceArray` have the same content but independent memory locations.

Note

The `arraycopy` method violates the Java naming convention. By convention, this method should be named `arrayCopy` (i.e., with an uppercase C).

7.13 Use the `arraycopy` method to copy the following array to a target array `t`:


```
int[] source = {3, 4, 5};
```

7.14 Once an array is created, its size cannot be changed. Does the following code resize the array?

```
int[] myList;
myList = new int[10];
// Sometime later you want to assign a new array to myList
myList = new int[20];
```

7.6 Passing Arrays to Methods

When passing an array to a method, the reference of the array is passed to the method.

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an `int` array:

```
public static void printArray(int[] array) {
  for (int i = 0; i < array.length; i++) {
    System.out.print(array[i] + " ");
  }
}
```

You can invoke it by passing an array. For example, the following statement invokes the `printArray` method to display `3`, `1`, `2`, `6`, `4`, and `2`.

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Note

The preceding statement creates an array using the following syntax:

```
new elementType[]{value0, value1, ..., valuek};
```

anonymous array

There is no explicit reference variable for the array. Such array is called an *anonymous array*.

pass-by-value

Java uses *pass-by-value* to pass arguments to a method. There are important differences between passing the values of variables of primitive data types and passing arrays.

- For an argument of a primitive type, the argument's value is passed.
- For an argument of an array type, the value of the argument is a reference to an array; this reference value is passed to the method. Semantically, it can be best described as *pass-by-sharing*, that is, the array in the method is the same as the array being passed. Thus, if you change the array in the method, you will see the change outside the method.

pass-by-sharing

Take the following code, for example:

```
public class Test {
  public static void main(String[] args) {
    int x = 1; // x represents an int value
    int[] y = new int[10]; // y represents an array of int values

    m(x, y); // Invoke m with arguments x and y

    System.out.println("x is " + x);
    System.out.println("y[0] is " + y[0]);
  }

  public static void m(int number, int[] numbers) {
    number = 1001; // Assign a new value to number
    numbers[0] = 5555; // Assign a new value to numbers[0]
  }
}
```

```
x is 1
y[0] is 5555
```

You may wonder why after `m` is invoked, `x` remains `1`, but `y[0]` become `5555`. This is because `y` and `numbers`, although they are independent variables, reference the same array, as illustrated in Figure 7.5. When `m(x, y)` is invoked, the values of `x` and `y` are passed to `number` and `numbers`. Since `y` contains the reference value to the array, `numbers` now contains the same reference value to the same array.

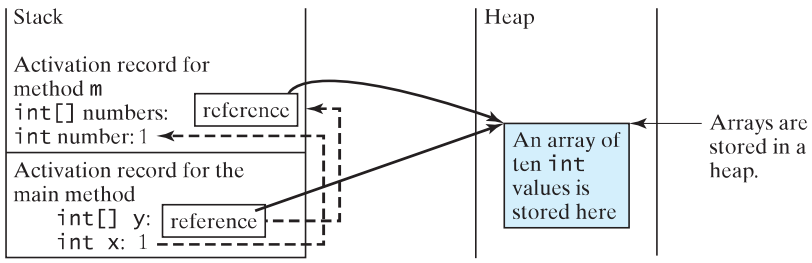


FIGURE 7.5 The primitive type value in `x` is passed to `number`, and the reference value in `y` is passed to `numbers`.

Note

Arrays are objects in Java (objects are introduced in Chapter 9). The JVM stores the objects in an area of memory called the *heap*, which is used for dynamic memory allocation.

heap

Listing 7.3 gives another program that shows the difference between passing a primitive data type value and an array reference variable to a method.

The program contains two methods for swapping elements in an array. The first method, named `swap`, fails to swap two `int` arguments. The second method, named `swapFirstTwoInArray`, successfully swaps the first two elements in the array argument.

LISTING 7.3 TestPassArray.java

```
1  public class TestPassArray {
2    /** Main method */
3    public static void main(String[] args) {
4      int[] a = {1, 2};
5
6      // Swap elements using the swap method
7      System.out.println("Before invoking swap");
8      System.out.println("array is {" + a[0] + ", " + a[1] + "}");
9      swap(a[0], a[1]);
10     System.out.println("After invoking swap");
11     System.out.println("array is {" + a[0] + ", " + a[1] + "}");
12
13     // Swap elements using the swapFirstTwoInArray method
14     System.out.println("Before invoking swapFirstTwoInArray");
15     System.out.println("array is {" + a[0] + ", " + a[1] + "}");
16     swapFirstTwoInArray(a);
17     System.out.println("After invoking swapFirstTwoInArray");
18     System.out.println("array is {" + a[0] + ", " + a[1] + "}");
19   }
20
21   /** Swap two variables */
22   public static void swap(int n1, int n2) {
23     int temp = n1;
24     n1 = n2;
25     n2 = temp;
26   }
27
28   /** Swap the first two elements in the array */
29   public static void swapFirstTwoInArray(int[] array) {
30     int temp = array[0];
31     array[0] = array[1];
32     array[1] = temp;
33   }
34 }
```

false swap

swap array elements

```
Before invoking swap
array is {1, 2}
After invoking swap
array is {1, 2}
Before invoking swapFirstTwoInArray
array is {1, 2}
After invoking swapFirstTwoInArray
array is {2, 1}
```

As shown in Figure 7.6, the two elements are not swapped using the `swap` method. However, they are swapped using the `swapFirstTwoInArray` method. Since the parameters in the `swap` method are primitive type, the values of `a[0]` and `a[1]` are passed to `n1` and `n2` inside the method when invoking `swap(a[0], a[1])`. The memory locations for `n1` and `n2` are independent of the ones for `a[0]` and `a[1]`. The contents of the array are not affected by this call.

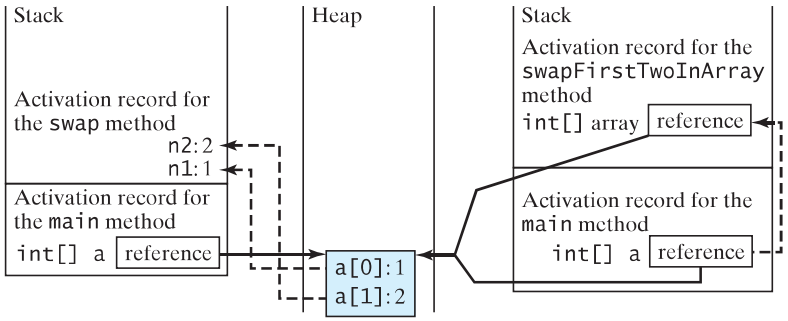


FIGURE 7.6 When passing an array to a method, the reference of the array is passed to the method.

The parameter in the `swapFirstTwoInArray` method is an array. As shown in Figure 7.6, the reference of the array is passed to the method. Thus the variables `a` (outside the method) and `array` (inside the method) both refer to the same array in the same memory location. Therefore, swapping `array[0]` with `array[1]` inside the method `swapFirstTwoInArray` is the same as swapping `a[0]` with `a[1]` outside of the method.

7.7 Returning an Array from a Method

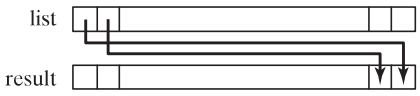
When a method returns an array, the reference of the array is returned.

You can pass arrays when invoking a method. A method may also return an array. For example, the following method returns an array that is the reversal of another array.

create array

return array

```
1  public static int[] reverse(int[] list) {
2    int[] result = new int[list.length];
3
4    for (int i = 0, j = result.length - 1;
5         i < list.length; i++, j--) {
6      result[j] = list[i];
7    }
8
9    return result;
10 }
```



Line 2 creates a new array `result`. Lines 4–7 copy elements from array `list` to array `result`. Line 9 returns the array. For example, the following statement returns a new array `list2` with elements `6`, `5`, `4`, `3`, `2`, `1`.

```
int[] list1 = {1, 2, 3, 4, 5, 6};
int[] list2 = reverse(list1);
```

7.15 Suppose the following code is written to reverse the contents in an array, explain why it is wrong. How do you fix it?


```
int[] list = {1, 2, 3, 5, 4};

for (int i = 0, j = list.length - 1; i < list.length; i++, j--) {
  // Swap list[i] with list[j]
  int temp = list[i];
  list[i] = list[j];
  list[j] = temp;
}
```

7.8 Case Study: Counting the Occurrences of Each Letter

This section presents a program to count the occurrences of each letter in an array of characters.

The program given in Listing 7.4 does the following:

1. Generates `100` lowercase letters randomly and assigns them to an array of characters, as shown in Figure 7.7a. You can obtain a random letter by using the `getRandomLowerCaseLetter()` method in the `RandomCharacter` class in Listing 6.10.
2. Count the occurrences of each letter in the array. To do so, create an array, say `counts`, of `26` `int` values, each of which counts the occurrences of a letter, as shown in Figure 7.7b. That is, `counts[0]` counts the number of `a`'s, `counts[1]` counts the number of `b`'s, and so on.

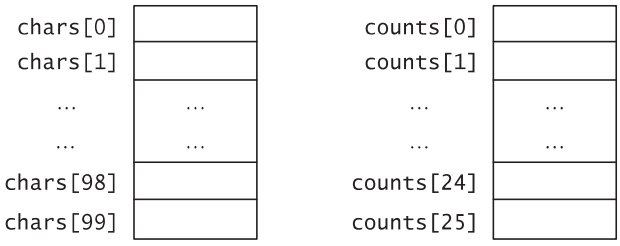


FIGURE 7.7 The `chars` array stores `100` characters, and the `counts` array stores `26` counts, each of which counts the occurrences of a letter.

LISTING 7.4 CountLettersInArray.java

```
1  public class CountLettersInArray {
2    /** Main method */
3    public static void main(String[] args) {
4      // Declare and create an array
5      char[] chars = createArray();                    create array
6
7      // Display the array
8      System.out.println("The lowercase letters are:");
9      displayArray(chars);                             pass array
10
```

return array

pass array

increase count

```
11      // Count the occurrences of each letter
12      int[] counts = countLetters(chars);
13
14      // Display counts
15      System.out.println();
16      System.out.println("The occurrences of each letter are:");
17      displayCounts(counts);
18    }
19
20    /** Create an array of characters */
21    public static char[] createArray() {
22      // Declare an array of characters and create it
23      char[] chars = new char[100];
24
25      // Create lowercase letters randomly and assign
26      // them to the array
27      for (int i = 0; i < chars.length; i++)
28        chars[i] = RandomCharacter.getRandomLowerCaseLetter();
29
30      // Return the array
31      return chars;
32    }
33
34    /** Display the array of characters */
35    public static void displayArray(char[] chars) {
36      // Display the characters in the array 20 on each line
37      for (int i = 0; i < chars.length; i++) {
38        if ((i + 1) % 20 == 0)
39          System.out.println(chars[i]);
40        else
41          System.out.print(chars[i] + " ");
42      }
43    }
44
45    /** Count the occurrences of each letter */
46    public static int[] countLetters(char[] chars) {
47      // Declare and create an array of 26 int
48      int[] counts = new int[26];
49
50      // For each lowercase letter in the array, count it
51      for (int i = 0; i < chars.length; i++)
52        counts[chars[i] - 'a']++;
53
54      return counts;
55    }
56
57    /** Display counts */
58    public static void displayCounts(int[] counts) {
59      for (int i = 0; i < counts.length; i++) {
60        if ((i + 1) % 10 == 0)
61          System.out.println(counts[i] + " " + (char)(i + 'a'));
62        else
63          System.out.print(counts[i] + " " + (char)(i + 'a') + " ");
64      }
65    }
66  }
```

```
The lowercase letters are:
e y l s r i b k j v j h a b z n w b t v
s c c k r d w a m p w v u n q a m p l o
a z g d e g f i n d x m z o u l o z j v
h w i w n t g x w c d o t x h y v z y z
q e a m f w p g u q t r e n n w f c r f

The occurrences of each letter are:
5 a 3 b 4 c 4 d 4 e 4 f 4 g 3 h 3 i 3 j
2 k 3 l 4 m 6 n 4 o 3 p 3 q 4 r 2 s 4 t
3 u 5 v 8 w 3 x 3 y 6 z
```

The `createArray` method (lines 21–32) generates an array of `100` random lowercase letters. Line 5 invokes the method and assigns the array to `chars`. What would be wrong if you rewrote the code as follows?

```
char[] chars = new char[100];
chars = createArray();
```

You would be creating two arrays. The first line would create an array by using `new char[100]`. The second line would create an array by invoking `createArray()` and assign the reference of the array to `chars`. The array created in the first line would be garbage because it is no longer referenced, and as mentioned earlier Java automatically collects garbage behind the scenes. Your program would compile and run correctly, but it would create an array unnecessarily.

Invoking `getRandomLowerCaseLetter()` (line 28) returns a random lowercase letter. This method is defined in the `RandomCharacter` class in Listing 6.10.

The `countLetters` method (lines 46–55) returns an array of `26 int` values, each of which stores the number of occurrences of a letter. The method processes each letter in the array and increases its count by one. A brute-force approach to count the occurrences of each letter might be as follows:

```
for (int i = 0; i < chars.length; i++)
  if (chars[i] == 'a')
    counts[0]++;
  else if (chars[i] == 'b')
    counts[1]++;
  ...
```

But a better solution is given in lines 51–52.

```
for (int i = 0; i < chars.length; i++)
  counts[chars[i] - 'a']++;
```

If the letter (`chars[i]`) is `a`, the corresponding count is `counts['a' - 'a']` (i.e., `counts[0]`). If the letter is `b`, the corresponding count is `counts['b' - 'a']` (i.e., `counts[1]`), since the Unicode of `b` is one more than that of `a`. If the letter is `z`, the corresponding count is `counts['z' - 'a']` (i.e., `counts[25]`), since the Unicode of `z` is `25` more than that of `a`.

Figure 7.8 shows the call stack and heap *during* and *after* executing `createArray`. See Checkpoint Question 7.18 to show the call stack and heap for other methods in the program.

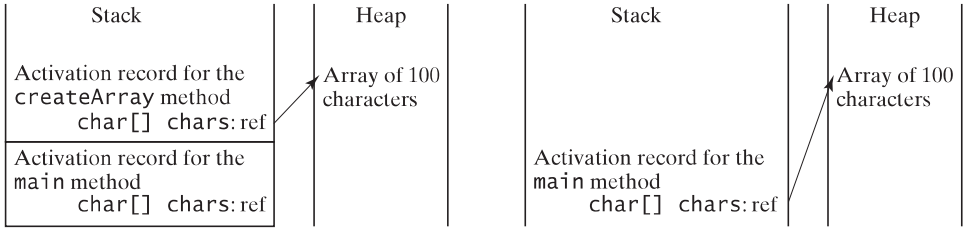


FIGURE 7.8 (a) An array of 100 characters is created when executing `createArray`. (b) This array is returned and assigned to the variable `chars` in the `main` method.

7.16 True or false? When an array is passed to a method, a new array is created and passed to the method.

7.17 Show the output of the following two programs:

```
public class Test {
  public static void main(String[] args) {
    int number = 0;
    int[] numbers = new int[1];

    m(number, numbers);

    System.out.println("number is " + number
      + " and numbers[0] is " + numbers[0]);
  }

  public static void m(int x, int[] y) {
    x = 3;
    y[0] = 3;
  }
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
    int[] list = {1, 2, 3, 4, 5};
    reverse(list);
    for (int i = 0; i < list.length; i++)
      System.out.print(list[i] + " ");
  }

  public static void reverse(int[] list) {
    int[] newList = new int[list.length];

    for (int i = 0; i < list.length; i++)
      newList[i] = list[list.length - 1 - i];

    list = newList;
  }
}
```

(b)

7.18 Where are the arrays stored during execution? Show the contents of the stack and heap during and after executing `displayArray`, `countLetters`, `displayCounts` in Listing 7.4.

7.9 Variable-Length Argument Lists

A variable number of arguments of the same type can be passed to a method and treated as an array.

You can pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows:

```
typeName... parameterName
```

In the method declaration, you specify the type followed by an ellipsis (`...`). Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

Java treats a variable-length parameter as an array. You can pass an array or a variable number of arguments to a variable-length parameter. When invoking a method with a variable number of arguments, Java creates an array and passes the arguments to it. Listing 7.5 contains a method that prints the maximum value in a list of an unspecified number of values.

LISTING 7.5 VarArgsDemo.java

```
1  public class VarArgsDemo {
2    public static void main(String[] args) {
3      printMax(34, 3, 3, 2, 56.5);
4      printMax(new double[]{1, 2, 3});
5    }
6
7    public static void printMax(double... numbers) {
8      if (numbers.length == 0) {
9        System.out.println("No argument passed");
10       return;
11     }
12
13     double result = numbers[0];
14
15     for (int i = 1; i < numbers.length; i++)
16       if (numbers[i] > result)
17         result = numbers[i];
18
19     System.out.println("The max value is " + result);
20   }
21 }
```

pass variable-length arg list

pass an array arg

a variable-length arg parameter

Line 3 invokes the `printMax` method with a variable-length argument list passed to the array `numbers`. If no arguments are passed, the length of the array is `0` (line 8).

Line 4 invokes the `printMax` method with an array.

7.19 What is wrong in the following method header?


```
public static void print(String... strings, double... numbers)
public static void print(double... numbers, String name)
public static double... print(double d1, double d2)
```

7.20 Can you invoke the `printMax` method in Listing 7.5 using the following statements?

```
printMax(1, 2, 2, 1, 4);
printMax(new double[]{1, 2, 3});
printMax(new int[]{1, 2, 3});
```

7.10 Searching Arrays

If an array is sorted, binary search is more efficient than linear search for finding an element in the array.

Searching is the process of looking for a specific element in an array—for example, discovering whether a certain score is included in a list of scores. Searching is a common task in computer programming. Many algorithms and data structures are devoted to searching. This section discusses two commonly used approaches, *linear search* and *binary search*.

linear search

binary search

7.10.1 The Linear Search Approach

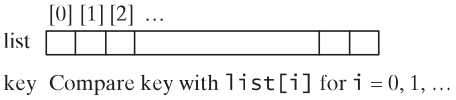
The linear search approach compares the key element `key` sequentially with each element in the array. It continues to do so until the key matches an element in the array or the array is exhausted without a match being found. If a match is made, the linear search returns the index

linear search animation on Companion Website

of the element in the array that matches the key. If no match is found, the search returns `-1`. The `linearSearch` method in Listing 7.6 gives the solution.

LISTING 7.6 LinearSearch.java

```
1 public class LinearSearch {
2   /** The method for finding a key in the list */
3   public static int linearSearch(int[] list, int key) {
4     for (int i = 0; i < list.length; i++) {
5       if (key == list[i])
6         return i;
7     }
8     return -1;
9   }
10 }
```



To better understand this method, trace it with the following statements:

```
int[] list = {1, 4, 4, 2, 5, -3, 6, 2};
int i = linearSearch(list, 4);  // Returns 1
int j = linearSearch(list, -4); // Returns -1
int k = linearSearch(list, -3); // Returns 5
```

The linear search method compares the key with each element in the array. The elements can be in any order. On average, the algorithm will have to examine half of the elements in an array before finding the key, if it exists. Since the execution time of a linear search increases linearly as the number of array elements increases, linear search is inefficient for a large array.

7.10.2 The Binary Search Approach

Binary search is the other common search approach for a list of values. For binary search to work, the elements in the array must already be ordered. Assume that the array is in ascending order. The binary search first compares the key with the element in the middle of the array. Consider the following three cases:

- If the key is less than the middle element, you need to continue to search for the key only in the first half of the array.
- If the key is equal to the middle element, the search ends with a match.
- If the key is greater than the middle element, you need to continue to search for the key only in the second half of the array.

Clearly, the binary search method eliminates at least half of the array after each comparison. Sometimes you eliminate half of the elements, and sometimes you eliminate half plus one. Suppose that the array has n elements. For convenience, let `n` be a power of `2`. After the first comparison, `n/2` elements are left for further search; after the second comparison, `(n/2)/2` elements are left. After the `k`th comparison, $n/2^k$ elements are left for further search. When $k = \log_2 n$, only one element is left in the array, and you need only one more comparison. Therefore, in the worst case when using the binary search approach, you need $\log_2 n+1$ comparisons to find an element in the sorted array. In the worst case for a list of `1024` (2^{10}) elements, binary search requires only `11` comparisons, whereas a linear search requires `1023` comparisons in the worst case.

binary search animation on Companion Website

The portion of the array being searched shrinks by half after each comparison. Let `low` and `high` denote, respectively, the first index and last index of the array that is currently being searched. Initially, `low` is `0` and `high` is `list.length-1`. Let `mid` denote the index of the middle element, so `mid` is `(low + high)/2`. Figure 7.9 shows how to find key `11` in the list {`2`, `4`, `7`, `10`, `11`, `45`, `50`, `59`, `60`, `66`, `69`, `70`, `79`} using binary search.

You now know how the binary search works. The next task is to implement it in Java. Don't rush to give a complete implementation. Implement it incrementally, one step at a time. You may start with the first iteration of the search, as shown in Figure 7.10a. It compares the key with the middle element in the list whose `low` index is `0` and `high` index is `list.length - 1`. If `key < list[mid]`, set the `high` index to `mid - 1`; if `key == list[mid]`, a match is found and return `mid`; if `key > list[mid]`, set the `low` index to `mid + 1`.

Next consider implementing the method to perform the search repeatedly by adding a loop, as shown in Figure 7.10b. The search ends if the key is found, or if the key is not found when `low > high`.

why not −1?

When the key is not found, `low` is the insertion point where a key would be inserted to maintain the order of the list. It is more useful to return the insertion point than `-1`. The method must return a negative value to indicate that the key is not in the list. Can it simply return `-low`? No. If the key is less than `list[0]`, `low` would be `0`. `-0` is `0`. This would indicate that the key matches `list[0]`. A good choice is to let the method return `-low - 1` if the key is not in the list. Returning `-low - 1` indicates not only that the key is not in the list, but also where the key would be inserted.

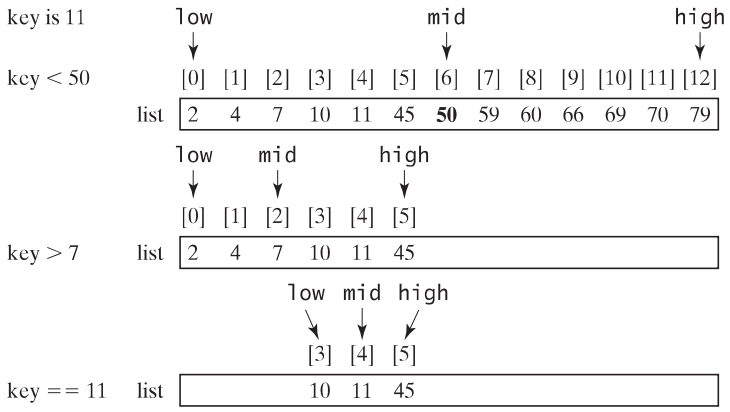


FIGURE 7.9 Binary search eliminates half of the list from further consideration after each comparison.

```
public static int binarySearch(
    int[] list, int key) {
  int low = 0;
  int high = list.length - 1;


    int mid = (low + high) / 2;
    if (key < list[mid])
      high = mid - 1;
    else if (key == list[mid])
      return mid;
    else
      low = mid + 1;


}
```

(a) Version 1

```
public static int binarySearch(
    int[] list, int key) {
  int low = 0;
  int high = list.length - 1;

  while (high >= low) {
    int mid = (low + high) / 2;
    if (key < list[mid])
      high = mid - 1;
    else if (key == list[mid])
      return mid;
    else
      low = mid + 1;
  }

  return -1; // Not found
}
```

(b) Version 2

FIGURE 7.10 Binary search is implemented incrementally.

The complete program is given in Listing 7.7.

LISTING 7.7 BinarySearch.java

```
1  public class BinarySearch {
2    /** Use binary search to find the key in the list */
3    public static int binarySearch(int[] list, int key) {
4      int low = 0;
5      int high = list.length - 1;
6
7      while (high >= low) {
8        int mid = (low + high) / 2;
9        if (key < list[mid])
10         high = mid - 1;
11       else if (key == list[mid])
12         return mid;
13       else
14         low = mid + 1;
15     }
16
17     return -low - 1; // Now high < low, key not found
18   }
19 }
```

first half

second half

The binary search returns the index of the search key if it is contained in the list (line 12). Otherwise, it returns `-low - 1` (line 17).

What would happen if we replaced `(high >= low)` in line 7 with (`high > low`)? The search would miss a possible matching element. Consider a list with just one element. The search would miss the element.

Does the method still work if there are duplicate elements in the list? Yes, as long as the elements are sorted in increasing order. The method returns the index of one of the matching elements if the element is in the list.

To better understand this method, trace it with the following statements and identify `low` and `high` when the method returns.

```
int[] list = {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79};
int i = BinarySearch.binarySearch(list, 2); // Returns 0
int j = BinarySearch.binarySearch(list, 11); // Returns 4
int k = BinarySearch.binarySearch(list, 12); // Returns -6
int l = BinarySearch.binarySearch(list, 1); // Returns -1
int m = BinarySearch.binarySearch(list, 3); // Returns -2
```

Here is the table that lists the `low` and `high` values when the method exits and the value returned from invoking the method.

Method	Low	High	Value Returned
`binarySearch(list, 2)`	0	1	0
`binarySearch(list, 11)`	3	5	4
`binarySearch(list, 12)`	5	4	-6
`binarySearch(list, 1)`	0	-1	-1
`binarySearch(list, 3)`	1	0	-2

Note

Linear search is useful for finding an element in a small array or an unsorted array, but it is inefficient for large arrays. Binary search is more efficient, but it requires that the array be presorted.

binary search benefits

7.21 If `high` is a very large integer such as the maximum `int` value `2147483647`, `(low + high) / 2` may cause overflow. How do you fix it to avoid overflow?

7.22 Use Figure 7.9 as an example to show how to apply the binary search approach to a search for key `10` and key `12` in list {`2`, `4`, `7`, `10`, `11`, `45`, `50`, `59`, `60`, `66`, `69`, `70`, `79`}.

7.23 If the binary search method returns `-4`, is the key in the list? Where should the key be inserted if you wish to insert the key into the list?

7.11 Sorting Arrays

Sorting, like searching, is a common task in computer programming. Many different algorithms have been developed for sorting. This section introduces an intuitive sorting algorithm: selection sort.

Suppose that you want to sort a list in ascending order. Selection sort finds the smallest number in the list and swaps it with the first element. It then finds the smallest number remaining and swaps it with the second element, and so on, until only a single number remains. Figure 7.11 shows how to sort the list {`2`, `9`, `5`, `4`, `8`, `1`, `6`} using selection sort.

VideoNote
Selection sort

selection sort

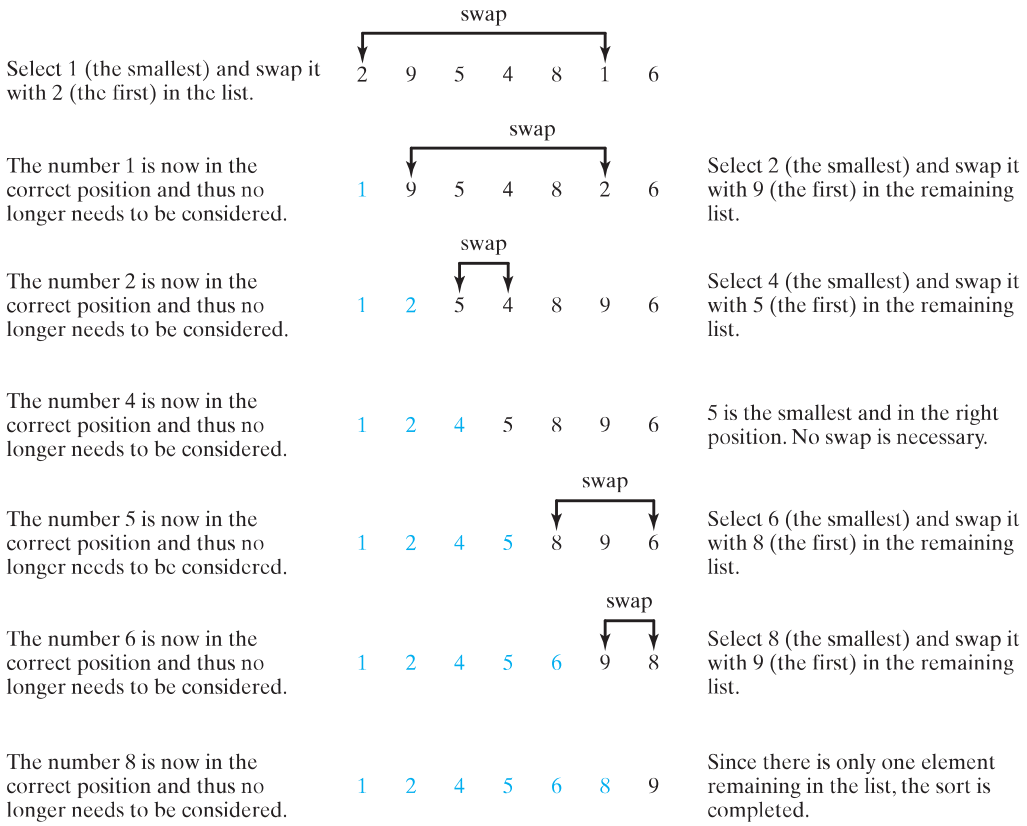


FIGURE 7.11 Selection sort repeatedly selects the smallest number and swaps it with the first number in the list.

You know how the selection-sort approach works. The task now is to implement it in Java. Beginners find it difficult to develop a complete solution on the first attempt. Start by writing the code for the first iteration to find the smallest element in the list and swap it with the first element, and then observe what would be different for the second iteration, the third, and so on. The insight this gives will enable you to write a loop that generalizes all the iterations.

selection sort animation on Companion Website

The solution can be described as follows:

```
for (int i = 0; i < list.length - 1; i++) {
  select the smallest element in list[i..list.length-1];
  swap the smallest with list[i], if necessary;
  // list[i] is in its correct position.
  // The next iteration applies on list[i+1..list.length-1]
}
```

Listing 7.8 implements the solution.

LISTING 7.8 SelectionSort.java

```
1  public class SelectionSort {
2    /** The method for sorting the numbers */
3    public static void selectionSort(double[] list) {
4      for (int i = 0; i < list.length - 1; i++) {
5        // Find the minimum in the list[i..list.length-1]
6        double currentMin = list[i];
7        int currentMinIndex = i;
8
9        for (int j = i + 1; j < list.length; j++) {
10         if (currentMin > list[j]) {
11           currentMin = list[j];
12           currentMinIndex = j;
13         }
14       }
15
16       // Swap list[i] with list[currentMinIndex] if necessary
17       if (currentMinIndex != i) {
18         list[currentMinIndex] = list[i];
19         list[i] = currentMin;
20       }
21     }
22   }
23 }
```

select

swap

The `selectionSort(double[] list)` method sorts any array of `double` elements. The method is implemented with a nested `for` loop. The outer loop (with the loop control variable `i`) (line 4) is iterated in order to find the smallest element in the list, which ranges from `list[i]` to `list[list.length-1]`, and exchange it with `list[i]`.

The variable `i` is initially `0`. After each iteration of the outer loop, `list[i]` is in the right place. Eventually, all the elements are put in the right place; therefore, the whole list is sorted.

To understand this method better, trace it with the following statements:

```
double[] list = {1, 9, 4.5, 6.6, 5.7, -4.5};
SelectionSort.selectionSort(list);
```

7.24 Use Figure 7.11 as an example to show how to apply the selection-sort approach to sort {`3.4`, `5`, `3`, `3.5`, `2.2`, `1.9`, `2`}.

7.25 How do you modify the `selectionSort` method in Listing 7.8 to sort numbers in decreasing order?

7.12 The `Arrays` Class

The `java.util.Arrays` class contains useful methods for common array operations such as sorting and searching.

The `java.util.Arrays` class contains various static methods for sorting and searching arrays, comparing arrays, filling array elements, and returning a string representation of the array. These methods are overloaded for all primitive types.

You can use the `sort` or `parallelSort` method to sort a whole array or a partial array. For example, the following code sorts an array of numbers and an array of characters.

`sort`
`parallelSort`

```
double[] numbers = {6.0, 4.4, 1.9, 2.9, 3.4, 3.5};
java.util.Arrays.sort(numbers); // Sort the whole array
java.util.Arrays.parallelSort(numbers); // Sort the whole array

char[] chars = {'a', 'A', '4', 'F', 'D', 'P'};
java.util.Arrays.sort(chars, 1, 3); // Sort part of the array
java.util.Arrays.parallelSort(chars, 1, 3); // Sort part of the array
```

Invoking `sort(numbers)` sorts the whole array `numbers`. Invoking `sort(chars, 1, 3)` sorts a partial array from `chars[1]` to `chars[3-1]`. `parallelSort` is more efficient if your computer has multiple processors.

You can use the `binarySearch` method to search for a key in an array. The array must be pre-sorted in increasing order. If the key is not in the array, the method returns `-(insertionIndex + 1)`. For example, the following code searches the keys in an array of integers and an array of characters.

`binarySearch`

```
int[] list = {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79};
System.out.println("1. Index is " +
       java.util.Arrays.binarySearch(list, 11));
System.out.println("2. Index is " +
       java.util.Arrays.binarySearch(list, 12));

char[] chars = {'a', 'c', 'g', 'x', 'y', 'z'};
System.out.println("3. Index is " +
       java.util.Arrays.binarySearch(chars, 'a'));
System.out.println("4. Index is " +
       java.util.Arrays.binarySearch(chars, 't'));
```

The output of the preceding code is

1. Index is 4
2. Index is −6
3. Index is 0
4. Index is −4

You can use the `equals` method to check whether two arrays are strictly equal. Two arrays are strictly equal if their corresponding elements are the same. In the following code, `list1` and `list2` are equal, but `list2` and `list3` are not.

`equals`

```
int[] list1 = {2, 4, 7, 10};
int[] list2 = {2, 4, 7, 10};
int[] list3 = {4, 2, 7, 10};
System.out.println(java.util.Arrays.equals(list1, list2)); // true
System.out.println(java.util.Arrays.equals(list2, list3)); // false
```

You can use the `fill` method to fill in all or part of the array. For example, the following code fills `list1` with `5` and fills `8` into elements `list2[1]` through `list2[5-1]`.

`fill`

```
int[] list1 = {2, 4, 7, 10};
int[] list2 = {2, 4, 7, 7, 7, 10};
java.util.Arrays.fill(list1, 5); // Fill 5 to the whole array
java.util.Arrays.fill(list2, 1, 5, 8); // Fill 8 to a partial array
```

toString

You can also use the `toString` method to return a string that represents all elements in the array. This is a quick and simple way to display all elements in the array. For example, the following code

```
int[] list = {2, 4, 7, 10};
System.out.println(Arrays.toString(list));
```

displays `[2, 4, 7, 10]`.

Check Point

7.26 What types of array can be sorted using the `java.util.Arrays.sort` method? Does this `sort` method create a new array?

7.27 To apply `java.util.Arrays.binarySearch(array, key)`, should the array be sorted in increasing order, in decreasing order, or neither?

7.28 Show the output of the following code:

```
int[] list1 = {2, 4, 7, 10};
java.util.Arrays.fill(list1, 7);
System.out.println(java.util.Arrays.toString(list1));

int[] list2 = {2, 4, 7, 10};
System.out.println(java.util.Arrays.toString(list2));
System.out.print(java.util.Arrays.equals(list1, list2));
```

7.13 Command-Line Arguments

Key Point

The `main` method can receive string arguments from the command line.

VideoNote
Command-line arguments

Perhaps you have already noticed the unusual header for the `main` method, which has the parameter `args` of `String[]` type. It is clear that `args` is an array of strings. The `main` method is just like a regular method with a parameter. You can call a regular method by passing actual parameters. Can you pass arguments to `main`? Yes, of course you can. In the following examples, the `main` method in class `TestMain` is invoked by a method in `A`.

```
public class A {
  public static void main(String[] args) {
    String[] strings = {"New York",
      "Boston", "Atlanta"};
    TestMain.main(strings);
  }
}
```

```
public class TestMain {
  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
}
```

A `main` method is just a regular method. Furthermore, you can pass arguments from the command line.

7.13.1 Passing Strings to the `main` Method

You can pass strings to a `main` method from the command line when you run the program. The following command line, for example, starts the program `TestMain` with three strings: `arg0`, `arg1`, and `arg2`:

```
java TestMain arg0 arg1 arg2
```

`arg0`, `arg1`, and `arg2` are strings, but they don't have to appear in double quotes on the command line. The strings are separated by a space. A string that contains a space must be enclosed in double quotes. Consider the following command line:

```
java TestMain "First num" alpha 53
```

It starts the program with three strings: `First num,` `alpha`, and `53`. Since `First num` is a string, it is enclosed in double quotes. Note that `53` is actually treated as a string. You can use `"53"` instead of `53` in the command line.

When the `main` method is invoked, the Java interpreter creates an array to hold the command-line arguments and pass the array reference to `args`. For example, if you invoke a program with `n` arguments, the Java interpreter creates an array like this one:

```
args = new String[n];
```

The Java interpreter then passes `args` to invoke the `main` method.

Note

If you run the program with no strings passed, the array is created with `new String[0]`. In this case, the array is empty with length `0`. `args` references to this empty array. Therefore, `args` is not `null`, but `args.length` is `0`.

7.13.2 Case Study: Calculator

VideoNote
Command-line argument

Suppose you are to develop a program that performs arithmetic operations on integers. The program receives an expression in one string argument. The expression consists of an integer followed by an operator and another integer. For example, to add two integers, use this command:

```
java Calculator 2 + 3
```

The program will display the following output:

```
2 + 3 = 5
```

Figure 7.12 shows sample runs of the program.

The strings passed to the main program are stored in `args`, which is an array of strings. The first string is stored in `args[0]`, and `args.length` is the number of strings passed.

Here are the steps in the program:

1. Use `args.length` to determine whether the expression has been provided as three arguments in the command line. If not, terminate the program using `System.exit(1)`.
2. Perform a binary arithmetic operation on the operands `args[0]` and `args[2]` using the operator in `args[1]`.

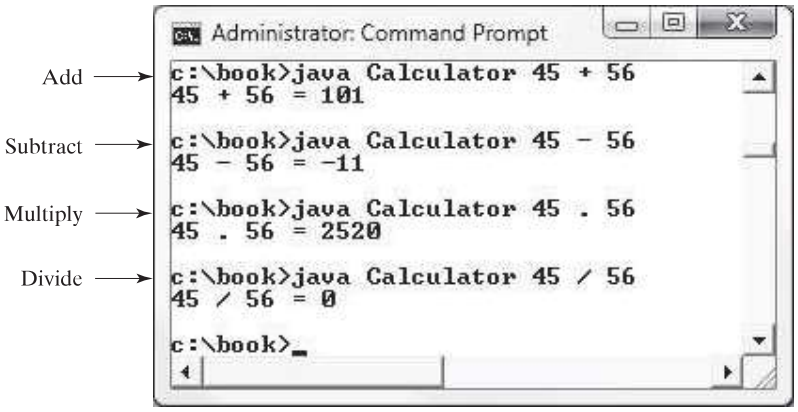


FIGURE 7.12 The program takes three arguments (`operand1 operator operand2`) from the command line and displays the expression and the result of the arithmetic operation.

The program is shown in Listing 7.9.

LISTING 7.9 Calculator.java

check argument

check operator

```
1  public class Calculator {
2    /** Main method */
3    public static void main(String[] args) {
4      // Check number of strings passed
5      if (args.length != 3) {
6        System.out.println(
7          "Usage: java Calculator operand1 operator operand2");
8        System.exit(0);
9      }
10
11     // The result of the operation
12     int result = 0;
13
14     // Determine the operator
15     switch (args[1].charAt(0)) {
16       case '+': result = Integer.parseInt(args[0]) +
17                          Integer.parseInt(args[2]);
18                 break;
19       case '-': result = Integer.parseInt(args[0]) -
20                          Integer.parseInt(args[2]);
21                 break;
22       case '.': result = Integer.parseInt(args[0]) *
23                          Integer.parseInt(args[2]);
24                 break;
25       case '/': result = Integer.parseInt(args[0]) /
26                          Integer.parseInt(args[2]);
27     }
28
29     // Display result
30     System.out.println(args[0] + ' ' + args[1] + ' ' + args[2]
31       + " = " + result);
32   }
33 }
```

`Integer.parseInt(args[0])` (line 16) converts a digital string into an integer. The string must consist of digits. If not, the program will terminate abnormally.

We used the `.` symbol for multiplication, not the common `*` symbol. The reason for this is that the `*` symbol refers to all the files in the current directory when it is used on a command line. The following program displays all the files in the current directory when issuing the command `java Test *`:

```
public class Test {
  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
}
```

To circumvent this problem, we will have to use a different symbol for the multiplication operator.

7.29 This book declares the `main` method as

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

Can it be replaced by one of the following lines?

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

```
public static void main(String x[])
static void main(String x[])
```

7.30 Show the output of the following program when invoked using

1. **java Test I have a dream**
2. **java Test "1 2 3"**
3. **java Test**

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Number of strings is " + args.length);
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
}
```

Key Terms

anonymous array 258
array 246
array initializer 248
binary search 265
garbage collection 256
index 246
indexed variable 248
linear search 265
off-by-one error 251
selection sort 269

Chapter Summary

1. A variable is declared as an *array* type using the syntax `elementType[] arrayRefVar` or `elementType arrayRefVar[]`. The style `elementType[] arrayRefVar` is preferred, although `elementType arrayRefVar[]` is legal.

2. Unlike declarations for primitive data type variables, the declaration of an array variable does not allocate any space in memory for the array. An array variable is not a primitive data type variable. An array variable contains a reference to an array.

3. You cannot assign elements to an array unless it has already been created. You can create an array by using the `new` operator with the following syntax: `new elementType[arraySize]`.

4. Each element in the array is represented using the syntax `arrayRefVar[index]`. An *index* must be an integer or an integer expression.

5. After an array is created, its size becomes permanent and can be obtained using `arrayRefVar.length`. Since the index of an array always begins with `0`, the last index is always `arrayRefVar.length - 1`. An out-of-bounds error will occur if you attempt to reference elements beyond the bounds of an array.

6. Programmers often mistakenly reference the first element in an array with index `1`, but it should be `0`. This is called the index *off-by-one error*.