Chapter 7

Single-Dimensional Arrays

7.1 Introduction

- Array is a data structure that stores a fixed-size sequential collection of elements of **the same types**.

7.2 Array Basics

- An array is used to store a collection of data, but it is often more useful to think of an array as **a collection of variables of the same type**.
- This section introduces how to declare array variables, create arrays, and process arrays

7.2.1 Declaring Array Variables

- Here is the syntax for declaring an array variable:

```
dataType[ ] arrayRefVar;
```

- The following code snippets are examples of this syntax:

```
double [ ] myList;
```

7.2.2 Creating Arrays

- Declaration of an array variable **doesn't** allocate any space in memory for the array.
- **Only** a storage location for the reference to an array is created.
- If a variable doesn't reference to an array, the value of the variable is ***null***.
- You can create an array by using the ***new*** operator with the following syntax:

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

- This element does two things:
  1) It creates an array using `new dataType[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 follows:

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

- Here is an example of such a statement

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

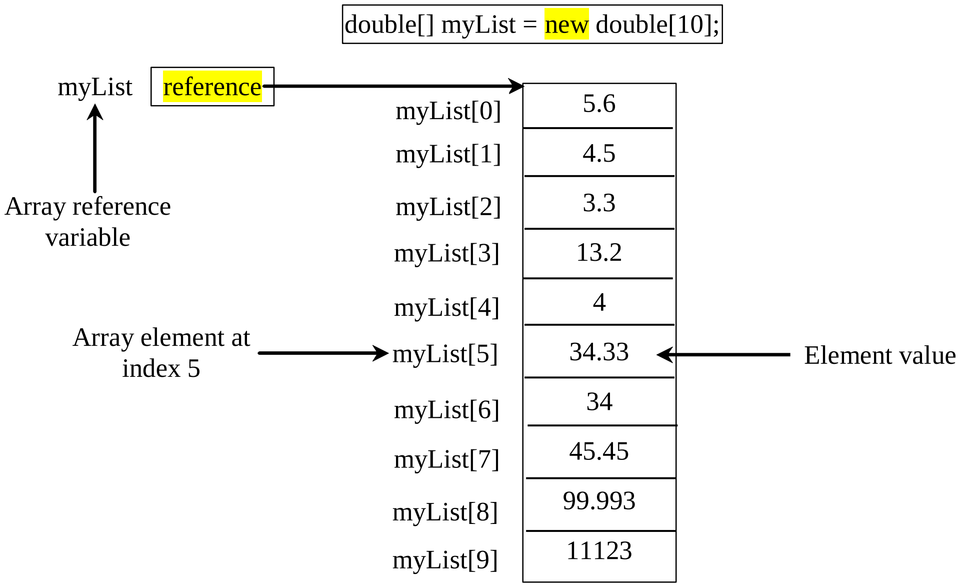

FIGURE 7.1 The array *myList* has ten elements of `double` type and `int` indices from 0 to 9.

- This statement declares an array variable, myList, creates an array of ten elements of double type, and assigns its reference to myList.

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**.

7.2.3 Array Size and Default values

- When space for an array is allocated, the array size must be given, to specify 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.
- 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

- The array elements are accessed through an index.
- The array indices are **0-based**, they start from 0 to arrayRefVar.*length-1*.
- In the example, myList holds ten double values and the indices from 0 to 9. The element *myList*[9] represents the last element in the array.
- After an array is created, an indexed variable can be used in the same way as a regular variable. For example:

```
myList[2] = myList[0] + myList[1];     //adds the values of the 1st and 2nd
                                       elements into the 3rd one

for (int i = 0; i < myList.length; i++) // the loop assigns 0 to myList[0]
     myList[i] = i;             //   1 to myList[1] .. and 9 to myList[9]
```

7.2.5 Array Initializers

- Java has a shorthand notation, known as the *array initializer* that combines declaring an array, creating an array and initializing it at the same time.

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

- This shorthand notation 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

- Using the shorthand notation, you have to declare, create, and initialize the array all in one statement. Splitting it would cause a syntax error. For example, the following 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. Here are the reasons why:
    1) All of the elements in an array are of the **same** type. They are evenly processed in the same fashion by repeatedly using a loop.
    2) Since the size of the array is **known**, it is natural to use `a for` loop.
- Here are some examples of processing arrays (Page 173):
    - (Initializing arrays)
    - (Printing arrays)
    - (Summing all elements)
    - (Finding the largest element)
    - (Finding the smallest index of the largest element)

7.2.7 Foreach Loops

- JDK 1.5 introduced a new for loop that enables you to traverse the complete array sequentially without using an index variable. For example, the following code displays all elements in the array myList:

```
for (double u: myList)
  System.out.println(u);
```

    - In general, the syntax is

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

    - 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.

7.3 Case Study: Analyzing Numbers

- Read the numbers of user inputs, compute their average, and find out how many numbers are above the average.

LISTING 7.1 AnalyzeNumbers.java

```
public class AnalyzeNumbers {
  public static void main(String[] args) {
    java.util.Scanner input = new java.util.Scanner(System.in);
    System.out.print("Enter the numbers of items: ");
    int n = input.nextInt();
    double[] numbers = new double[n];
    double sum = 0;

    System.out.print("Enter the numbers: ");
    for (int i = 0; i < n; i++) {
      numbers[i] = input.nextDouble();
      sum += numbers[i];
    }

    double average = sum / n;

    int count = 0; // The numbers of elements above average
    for (int i = 0; i < n; i++)
      if (numbers[i] > average)
        count++;

    System.out.println("Average is " + average);
    System.out.println("Number of elements above the average is "
      + count);
  }
}
```

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

7.4 Case Study: Deck of Cards

- The problem is to write a program that picks **four** cards **randomly** from a deck of 52 cards. All the cards can be represented using an array named deck, filled with initial values 0 to 52, as follows:

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

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

LISTING 7.2 DeckOfCards.java

```
public class DeckOfCards {
  public static void main(String[] args) {
    int[] deck = new int[52];
    String[] suits = {"Spades", "Hearts", "Diamonds", "Clubs"};
    String[] ranks = {"Ace", "2", "3", "4", "5", "6", "7", "8", "9",
      "10", "Jack", "Queen", "King"};

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

    // Shuffle the cards
    for (int i = 0; i < deck.length; i++) {
      // Generate an index randomly
      int index = (int)(Math.random() * deck.length);
      int temp = deck[i];
      deck[i] = deck[index];
     deck[index] = temp;
    }

    // Display the first four cards
    for (int i = 0; i < 4; i++) {
      String suit = suits[deck[i] / 13];
      String rank = ranks[deck[i] % 13];
      System.out.println("Card number " + deck[i] + ": "
        + rank + " of " + suit);
    }
  }
}
```

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

7.5 Copying Arrays

- 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;
```

- This statement does **not** copy the contents of the array referenced by *list1* to *list2*, but merely **copies the reference value** from *list1* to *list2*. After this statement, *list1* and *list2* reference to the same array, as shown below.

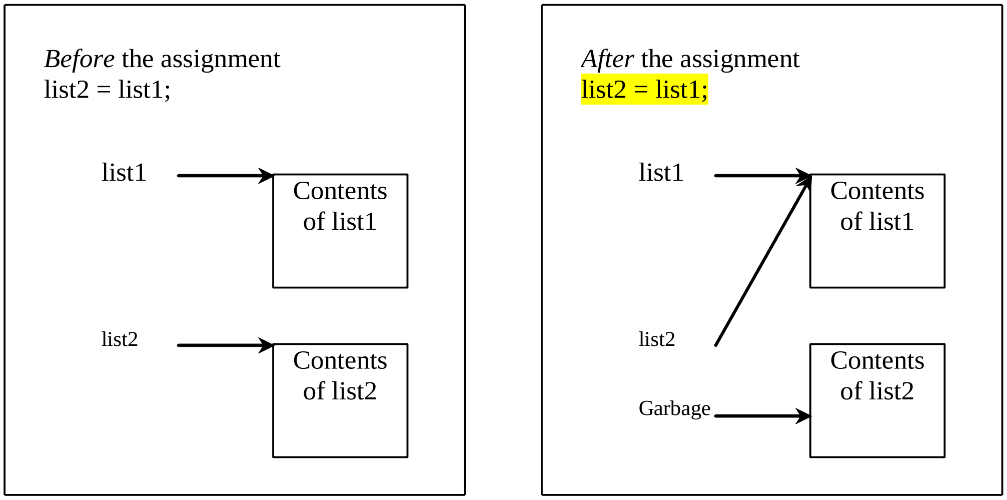

FIGURE 7.4 Before the assignment, list1 and list2 point to separate memory locations. After the assignments the reference of the list1 array is passed to list2

- The ***array previously referenced by*** **list2** ***is no longer referenced; it becomes*** *garbage*, ***which will be automatically collected by the Java Virtual Machine.***
- You can use assignment statements to copy primitive data type variables, but not arrays.
- Assigning one array variable to another variable actually copies one reference to another and makes both variables point to the same **same memory location**.

- There are three ways to copy arrays:
    - Use a **loop** to copy individual elements.
    - Use the static ***arraycopy*** method in the *System* class.
    - Use the **clone** method to copy arrays. “Introduced in chapter 9.”

- Using a **loop**:

```
int[] sourceArray = {2, 3, 1, 5, 10};
int[] targetArray = new int[sourceArray.length];

for (int i = 0; i < sourceArrays.length; i++)
   targetArray[i] = sourceArray[i];
```

- The **arraycopy** method:

```
arraycopy(sourceArray, src_pos, targetArray, tar_pos, length);
```

Example:

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

- The number of elements copied from `sourceArray` to `targetArray` is indicated by length.
- The `arraycopy` 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 take place, `targetArray` and `sourceArray` have the same content but independent memory locations.

7.6 Passing Arrays to Methods

- The following method displays the elements of an int array:

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

The following invokes the method to display 3, 1, 2, 6, 4, and 2.

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

printArray(new int[]{3, 1, 2, 6, 4, 2});
      // anonymous array; no explicit reference variable for the array
```

- 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 an argument contains a reference to an array; this **reference** is passed to the method.

```
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
```

- *y* and *numbers* reference to the same array, although *y* and *numbers* are independent variables.
- When invoking *m(x, y)*, 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.