Chapter 7

Single-Dimensional Arrays

Objectives

- To declare array reference variables and create arrays
- To obtain array size
- To access array elements using indexes
- To declare, create, and initialize an array using an array initializer
- Using the for each loops
- To copy contents from one array to another
- To define and invoke methods with array arguments and return values
- To define a method with a variable-length argument list
- To search elements using the linear search algorithm.
- To use the methods in the **java.util.Arrays** class

Arrays: An Introduction

- A *data structure* is a way to organize data in a computer.
- A *one-dimensional array* is a data structure that stores a fixed-size *sequence* of values, all of the same type.
- We refer to the components of an array as its elements.
- We use *indexing* to refer to the array elements: If we have n elements in an array, we think of the elements as being numbered from 0 to n-1.
- So, 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.

Step 1: Declaring Array Reference Variables

- To declare an array, specify
  - Array variable name
  - the array's *element type*
- The syntax for declaring an array variable: elementType[] arrayRefVar; For example,

```
double[] myList;
```

Step 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.
- To assign elements to an array, you have to create it by specifying the length of the array, i.e., the number of elements in the array.
- To create an array after it has been declared by using the **new** operator and assign its reference to the variable with the following syntax:

  arrayRefVar = **new** elementType[length];
- E.g. myList = new double[10];

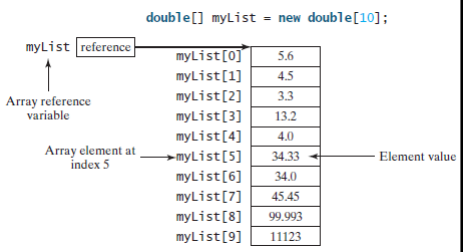
Declaring & Creating and Assigning can be Combined

- 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**];
- When an array is created, its elements are assigned the default value of **0** for the numeric primitive data types, **\u0000 (NULL character)** for **char** types, and **false** for **boolean** types.
- To assign values to the elements, use the syntax: arrayRefVar[index] = value;

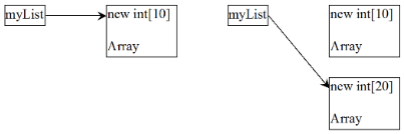
  E.g. myList[0] = 5.6;



Array Size

- 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. What actually happens with the following code is that the second assignment statement myList = new int[20] creates a new array and assigns its reference to myList.

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



- A work around is to create a new array of the desired size, and copy the contents from the original array to the new array, using java.lang.System.arraycopy(...); which will be covered on slide 18.
- Size can be obtained using **arrayRefVar.length**. For example, **myList.length** is **10**.

length vs length()

- **array.length :** length is a final variable applicable for arrays. With the help of length variable, we can obtain the size of the array.
- **string.length() :** length() method returns the number of characters presents in the string.
- **length vs length()**
  - The length variable is applicable to array but not for string objects whereas the length() method is applicable for string objects but not for arrays.
  - **length** can be used for int[], double[], String[] // to know the length of the arrays.
  - **length()** can be used for String // String class related Objects for the length of the String
- To directly accesses a field member of array we can use **.length;** whereas **.length()** invokes a method to access a field member.

length vs length() : an example

```
public class Test {
public static void main(String[] args)
  {
    // Here array is the array name of int type
    int[] array = new int[4];
    System.out.println("The size of the array is " + array.length);   // 4

    // Here str is a string object
    String str = “Welcome";
    System.out.println("The size of the String is " + str.length()); //7
  }
}
```


Accessing Array Elements

- The array elements are accessed through the index.
- Array indices are **0** based; that is, they range from **0** to **arrayRefVar.length-1**.
- Each element in the array is represented using the following syntax, known as an *indexed variable*:

  arrayRefVar[index];
- 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**];

Accessing Array Elements

- 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;
}
```

- 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**.

Array Initializers

- *Array initializer* combines the declaration, creation, and initialization of an array in one statement using the following syntax:

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

- For example, the following statement declares, creates, and initializes the array **myList** with four elements,

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

```
double[] myList = new double[4];
myList[0] = 1.9;
myList[1] = 2.9;
myList[2] = 3.4;
myList[3] = 3.5;
```

- The **new** operator is not used in the array-initializer syntax. Splitting it would cause a syntax error. Thus, the next statement is wrong:

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

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();
```


Processing Arrays (cont'd)

2. *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] + " ");
}
```

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);
```

The above method ONLY works for array of the **char[]** type.

```
int[] city1 = {1, 2, 3, 4};
System.out.println(city1);    // [I@15db9742
System.out.println(Arrays.toString(city1));    //  output [1, 2, 3, 4]  -- to be discussed in slide 38
```

You can't decipher anything until you are quite familiar of this array format, and even then it doesn't tell anything about contents of array. It just prints type of element and hashcode.

You have to use import java.util.Arrays;

Processing Arrays (cont'd)

3. *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];
}
```

4. *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**.

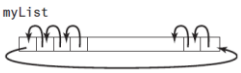
```
double max = myList[0];  // to begin with, assume the first element is the largest
int indexOfMax = 0;
for (int i = 1; i < myList.length; i++) {
  if (myList[i] > max) {
      max = myList[i];  // use variable max to hold the value of the current largest element
      indexOfMax = i;  // use variable indexOfMax to store the smallest index of the largest element
  }
}
```

Suppose that the array **myList** is {**1**, **5**, **3**, **4**, **5**, **5**}. The largest element is **5**, and the smallest index for **5** is **1**.

Processing Arrays (cont'd)

5. *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;
```



Foreach Loops

- The *forEach* loop provides programmers with **a new way for iterating over a collection**.
- A *foreach loop* 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);
}
```

```
for (int e = 0; e < myList.length; e++) {
  System.out.println(myList[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**.
- 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**.
- In general, the syntax for a foreach loop is

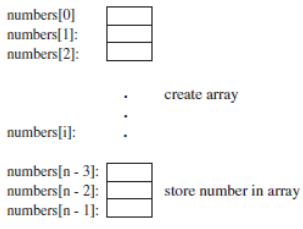
```
for (elementType element: arrayRefVar) {
  // Process the element
}
```


Case Study: Analyzing Numbers

- *The problem is to write a program that finds the number of items above the average of all items.*
- Read the case study (7.4) on Deck of Cards

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 }
```



Copying Arrays: common error

- *To copy the contents of one array into another, you have to copy the array's individual elements into the other array.*
- You could attempt to use the assignment statement (=), list2 = list1; 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*).

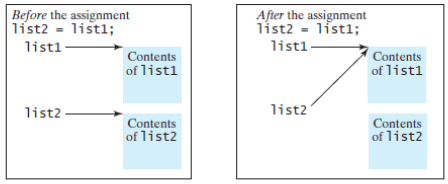


FIGURE 7.4 Before the assignment statement, `list1` and `list2` point to separate memory locations. After the assignment, the reference of the `list1` array is passed to `list2`.

Copying Arrays

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

Copying Arrays: Using a loop

- 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];
}
```


Copying Arrays: **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.
- Did you notice that the **arraycopy** method violates the Java naming convention?

Passing Arrays to Methods

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

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

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


Passing Arguments by Values

- 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.

Passing Arguments by Values: an example

- 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.
- 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.
- Read **LISTING 7.3** TestPassArray.java for another example.

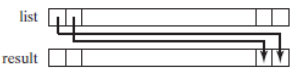
```
public class TestArrayArguments {
  public static void main(String[] args) {
    int x = 1; // x represents an int value
    int[] y = new int[10]; // y represents an array of int values

    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
```


Returning an Array from a Method

- *When a method returns an array, the reference of the array is returned.*
- For example, the following method returns an array that is the reversal of another 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 }
```



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);

- Read the case study (7.8) on "Counting the Occurrences of Each Letter"

Variable-Length Argument Lists

- *A variable number of arguments of the same type can be passed to a method and treated as an array.*
- In the method declaration, you specify the type followed by an ellipsis (**...**). The parameter in the method is declared as follows:

  typeName... parameterName
- 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.
- 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.

Variable-Length Argument Lists: an example

- Listing 7.5 contains a method that prints the maximum value in a list of an unspecified number of values.
- 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.

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 }
```

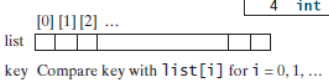
Searching Arrays: The Linear Search Approach

- 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 of the element in the array that matches the key.
- If no match is found, the search returns **-1**.
- 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.

The Linear Search Approach: implementation

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:

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

- The linear search method compares the key with each element in the array.
- 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.

The **Arrays** Class: sort method

- 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** method to sort a whole array or a partial array. For example, the following code sorts an array of characters.

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

  Invoking **sort(chars, 1, 3)** sorts a partial array from **chars[1]** to **chars[3-1]**.
- You can sort an array of any primitive types except boolean. The sort method is void, so it does not return a new array.

The **Arrays** Class: equals method

- 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.

```
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
```

The **Arrays** Class: **toString** method

- 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 displays **[2, 4, 7, 10]**.

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

Need to: import java.util.Arrays;

Chapter Summary

- A variable is declared as an *array* type using the syntax **elementType[] arrayRefVar** or **elementType arrayRefVar[]**. The style **elementType[] arrayRefVar** is preferred.
- 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.
- 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]**.
- Each element in the array is represented using the syntax **arrayRefVar[index]**. An *index* must be an integer or an integer expression.
- 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.

Chapter Summary

- 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.
- Java has a shorthand notation, known as the *array initializer*, which combines declaring an array, creating an array, and initializing an array in one statement, using the syntax
  elementType[] arrayRefVar = {value0, value1, ..., value*k*}
- When you pass an array argument to a method, you are actually passing the reference of the array; that is, the called method can modify the elements in the caller's original array.

Exercises

- Write the code to declare a variable **myList** that references an array of double elements.
- Write the code to create an array that can store 5 elements and assign its reference to the variable myList declared in the previous question.
- What is the representation of the third element in the array declared?
- Write the code to use an array initializer to create another array with the initial values 1.9, 2.9, 3.4 and 3.5.
- What is the result of this program?

```
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;
  }
}
```

Ideas for further practice

- Read 7.4 Case Study: Deck of Cards (as indicated on slide 17)
- Read **LISTING 7.3** TestPassArray.java as an example. (as indicated on slide 24)
- Read 7.8 Case Study: Counting the Occurrences of Each Letter (as indicated on slide 25)