**CHAPTER 7: SINGLE-DIMENSIONAL ARRAYS**

* 1. **INTRODUCTION**

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

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.

* 1. **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.

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.

**DECLARING ARRAY BARIABLES**

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.

**CREATING ARRAYS**

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 array variable is declared, you can create an array by using the new operator and assign its reference to the variable.

Here is the syntax:

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

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];**

Here is an example of such a statement:

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

This statement declares an array variable, **myList**, creates an array of 10 elements of **double** type, and assigns its reference to **myList**.

To assign values to the elements, use the syntax:

**arrayRefVar[index] = value;**

**ARRAY SIZE AND DEFAULT VALUES**

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**.

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.

**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];**

**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};**

**PROCESSING ARRAYS**

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

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

**Example of processing arrays:**

Assume that the array is created as follows:

double[] myList = new double[10];

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;

}

**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.

the syntax for a foreach loop is:

for (elementType element: arrayRefVar) {

// Process the element

}

* 1. **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;**

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.

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.

(Using loop) 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];

}

Another approach is to use the **arraycopy** method 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.

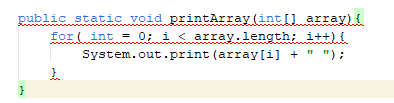
The number of elements copied from sourceArray to targetArray is indicated by length.

* 1. **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:



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});

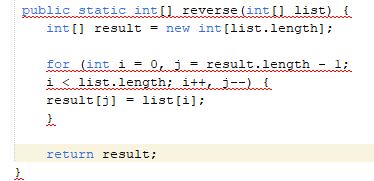
Java uses **pass-by-value** to pass arguments to a method. There are important differences between passing the values of variables of primitive data types and passing arrays.

* 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.
  1. **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:

****

* 1. **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.

* 1. **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.

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**.

**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 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.

**THE BINARY SEARCH APPROACH**

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.

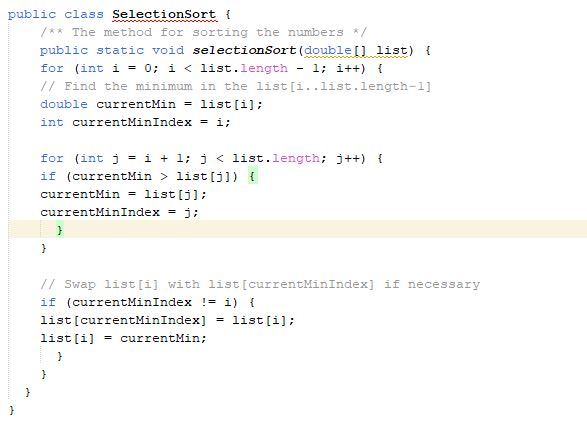
Consider the following three cases:

1. 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.
2. If the key is equal to the middle element, the search ends with a match.
3. 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.

The binary search method eliminates at least half of the array after each comparison.

* 1. **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.

****

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** is iterated in order to find the smallest element in the list, which ranges from **list[i]** to **list[list.length−1],** and exchanges 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.

* 1. **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.

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).**

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.

* 1. **COMMAND-LINE ARGUMENTS**

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

The **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.

Furthermore, you can pass arguments to a main method from the command line.

**PASSING STRINGS TO THE MAIN METHOD**

You can pass strings to a **main** method from the command line when you run the program.

For example, the following command line 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 END!**