Chapter 7

Single-Dimensional Arrays

Objectives

- To declare array reference variables and create arrays
- To obtain array size using **arrayRefVar.length** and know default values in an array
- To access array elements using indexes
- To declare, create, and initialize an array using an array initializer
- To simplify programming using the for each loops
- To copy contents from one array to another
- To develop 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 or binary search algorithm.
- To sort an array using the selection sort approach
- To use the methods in the **java.util.Arrays** class
- To pass arguments to the main method from the command line

Arrays: An Introduction

- *A single array variable can reference a large collection of data.*
- Java provides a data structure, the *array*, which stores a fixed-size sequential collection of elements of the same type.
- For example, you can store 100 numbers into an array and access them through a single array variable.
- *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.
- 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 Variables

- 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;
```

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. 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 with the following syntax:

  arrayRefVar = **new** elementType[arraySize];
- This statement does two things:

  (1) it creates an array using **new elementType[arraySize]**;

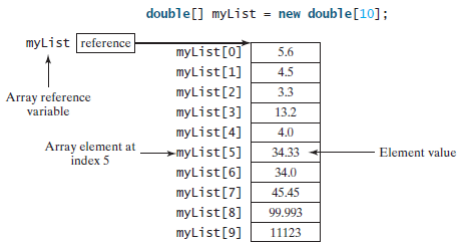
  (2) it assigns the reference of the newly created array to the variable **arrayRefVar**.

All in one: 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**];
- This statement declares an array variable, **myList**, creates an array of ten elements of **double** type, and assigns its reference to **myList**.



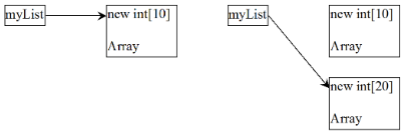
- When an array is created, its elements are assigned the default value of **0** for the numeric primitive data types, **\u0000** for **char** types, and **false** for **boolean** types.
- To assign values to the elements, use the syntax:

  arrayRefVar[index] = value;

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**.

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];
```

- 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, ..., value*k*};

- 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**};

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

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);
```


Processing Arrays (cont'd)

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];
}
```

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

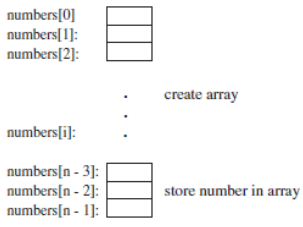
```
for (elementType element: arrayRefVar) {
  // Process the element
}
```


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 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*).

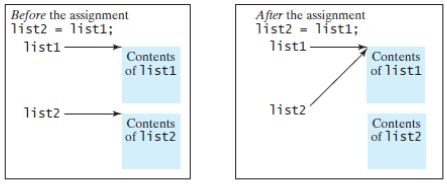


FIGURE 7.4 Before the assignment statement, `list1` and `list2` point to separate memory locations. After the assignment, the reference of the `list1` array is passed to `list2`.

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.

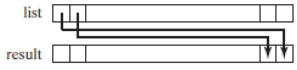
```
public class Test {
  public static void main(String[] args) {
    int x = 1; // x represents an int value
    int[] y = new int[10]; // y represents an array of int values

    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
```


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 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 }
```

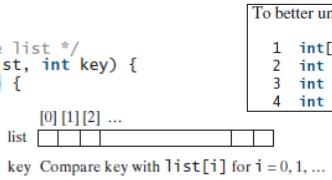
Searching Arrays: The Linear Search Approach

- 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.

Searching Arrays: 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:
  - 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.

The Binary Search Approach (cont'd)

- Clearly, the binary search method eliminates at least half of the array after each comparison.

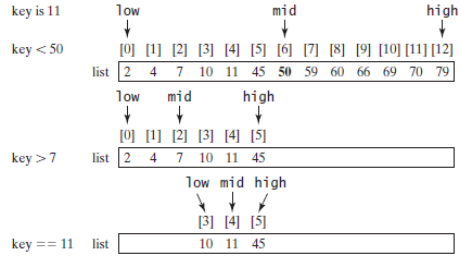


FIGURE 7.9 Binary search eliminates half of the list from further consideration after each comparison.

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

The Binary Search Approach: implementation

- 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.

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 }
```


The Binary Search Approach: implementation

- 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

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 }
```

Sorting Arrays: 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.

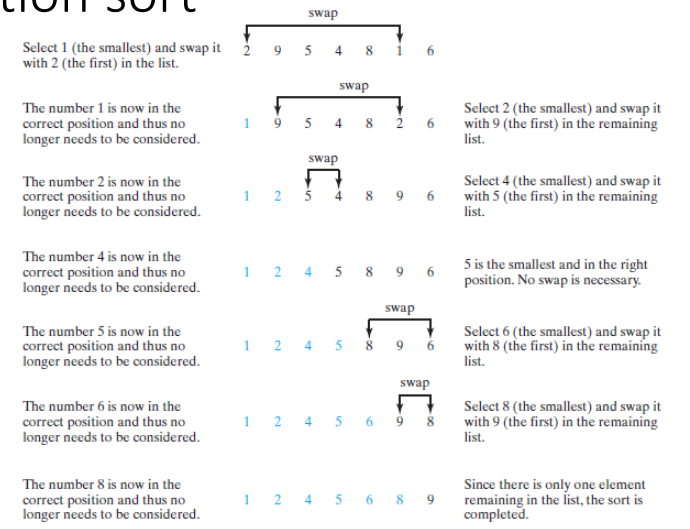


FIGURE 7.11 Selection sort repeatedly selects the smallest number and swaps it with the first number in the list.

Selection sort: an implementation

- 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);
```

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 }
```

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]
}
```

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** 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.

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

The **Arrays** Class: binearySearch method

- You can use the **binarySearch** method to search for a key in an array. The array must be presorted 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.

```
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

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: fill method

- 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]**.

```
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
```

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));
```


Passing arguments to the **main** method

- The header for the **main** method 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]);
  }
}
```

Command-Line Arguments for the **main** Method

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

Note that **53** is actually treated as a string. You can use **"53"** instead of **53** in the command line.

Command-Line Arguments for the **main** Method (cont'd)

- 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.
- 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**.

Case Study: Calculator

- 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.

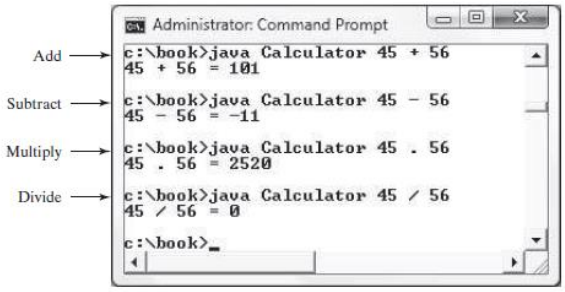


FIGURE 7.12 The program takes three arguments (operand1 operator operand2) from the command line and displays the expression and the result of the arithmetic operation.

Case Study: Calculator – implementation

- **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.

LISTING 7.9 Calculator.java

```
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 }
```

Chapter Summary

- A variable is declared as an *array* type using the syntax **elementType[] arrayRefVar** or **elementType arrayRefVar[]**. The style **elementType[] arrayRefVar** is preferred.
- 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.
- 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.
- If an array is sorted, *binary search* is more efficient than *linear search* for finding an element in the array.
- *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 first element in the remaining list, and so on, until only a single number remains.

Ideas for further practice

- Read 7.4 Case Study: Deck of Cards (as indicated on slide 14)
- Read **LISTING 7.3** TestPassArray.java as an example. (as indicated on slide 21)
- Read 7.8 Case Study: Counting the Occurrences of Each Letter (as indicated on slide 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**}.
- 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**}.