Single-Dimensional Arrays

Introduction

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

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

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;
```

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];
```

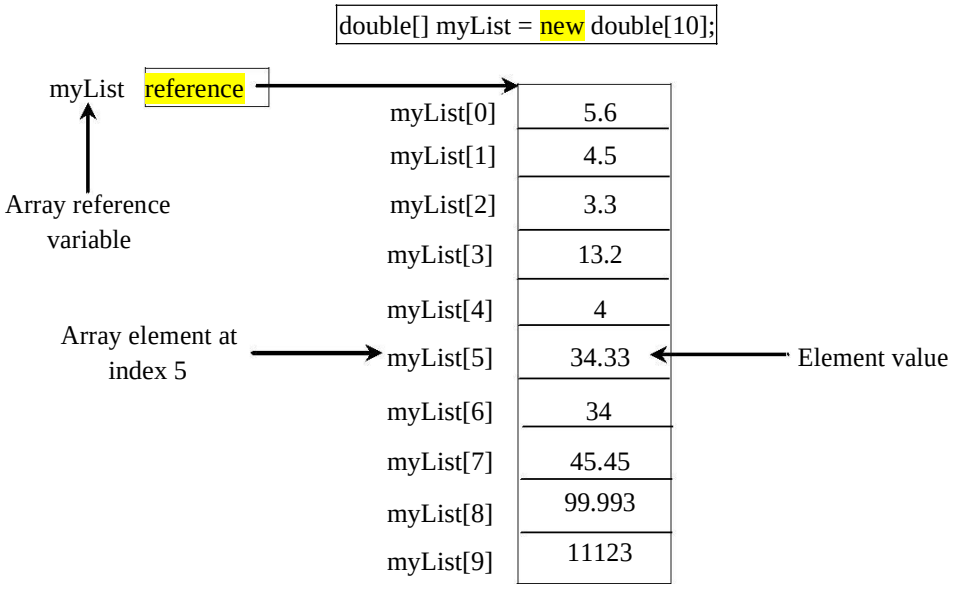


FIGURE : The array *myList* has ten elements of `double` type and `int` indices from 0 to 9.

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**.

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.

Array Indexed Variables

- 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]
```

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};
```

Processing Arrays

- When processing array elements, you will often use a ***for*** loop. Here are the reasons why:
- 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.
- Since the size of the array is **known**, it is natural to use `a for` loop.

- Here are some examples of processing arrays
  - (Initializing arrays)
  - (Printing arrays)
  - (Summing all elements)
  - (Finding the largest element)
  - (Finding the smallest index of the largest element)

For-each 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.

Program:

Suppose you play the Pick-10 lotto. Each ticket has 10 unique numbers ranging from 1 to 99. You buy a lot of tickets. You like to have your all tickets to **cover** all numbers from 1 to 99. Write a program that reads the ticket numbers from a file and checks whether all numbers are covered. Assume the last number in the file is ***0***.

LottoNumbers

```
import java.util.Scanner;

public class LottoNumbers
{
  public static void main(String args[])
  {
    Scanner input = new Scanner(System.in);
    boolean[] isCovered = new boolean[100]; // default false

    // Read all numbers and mark corresponding element covered
    int number = input.nextInt();
    while (number != 0) {
      isCovered[number] = true;
      number = input.nextInt();
    }

     //Check if all covered
    boolean allCovered = true; // Assume all covered
    for (int i = 1; i <= 99; i++)
      if (!isCovered[i]) {
        allCovered = false; // Find one number is not covered
        break;
      }
```

```
    //Display result
    if (allCovered)  //allCovered == true
      System.out.println("The tickets cover all numbers");
      else
      System.out.println("The tickets don't cover all numbers");
  }
}
```

```
1 3 87 62 30 90 10 21 46 27
12 40 83 9 39 88 95 59 20 37
80 40 87 67 31 90 11 24 56 77
11 48 51 42 8 74 1 41 36 53
52 82 16 72 19 70 44 56 29 33
54 64 99 14 23 22 94 79 55 2
60 86 34 4 31 63 84 89 7 78
43 93 97 45 25 38 28 26 85 49
47 65 57 67 73 69 32 71 24 66
92 98 96 77 6 75 17 61 58 13
35 81 18 15 5 68 91 50 76
0
```

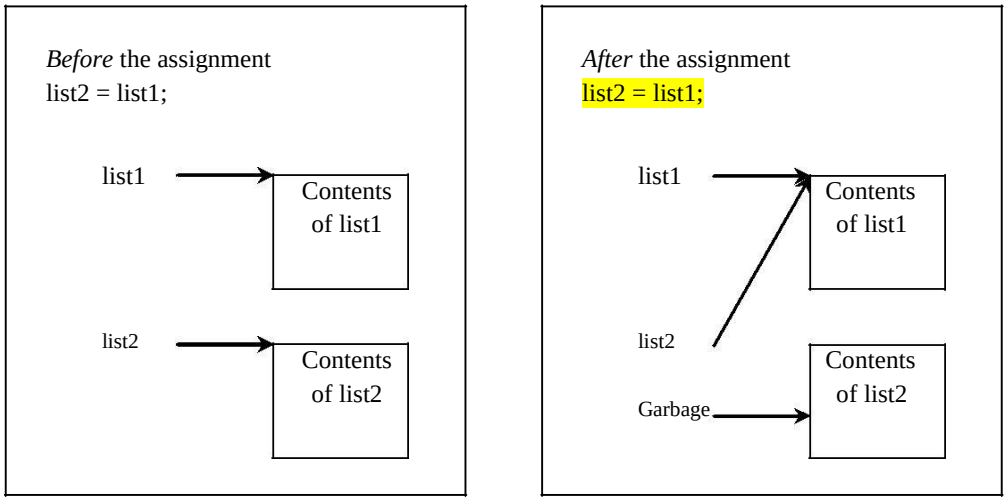
The tickets cover all numbers

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.



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 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.

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.

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.

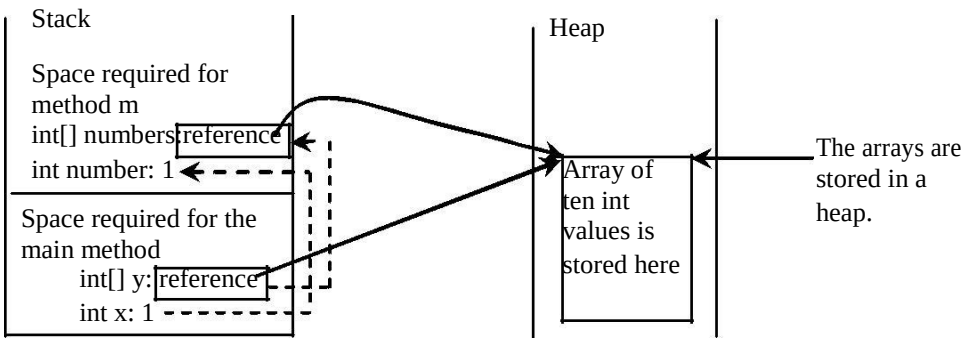
```
public class Test {
  public static void main(String[] args) {
    int x = 1; // x represents an int value
    int[] y = new int[10]; // y represents an array of int values

    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.

The JVM stores the array in an area of memory called ***heap***, which is used by dynamic memory allocation where blocks of memory are allocated and freed in an arbitrary order.



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

Returning an Array from a Method

You can pass arrays to invoke a method. A method may also return an array.
For example, the method below returns an array that is the reversal of another array:

```
public static int [] reverse(int[] list)
{
  int[] result = new int[list.length];       // creates new array result
  for (int i = 0, j = result.length - 1; //  copies elements from array
       i < list.length; i++, j--) {       // list to array result
    result[j] = list[i];
  }
  return result;
}
```

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);
```