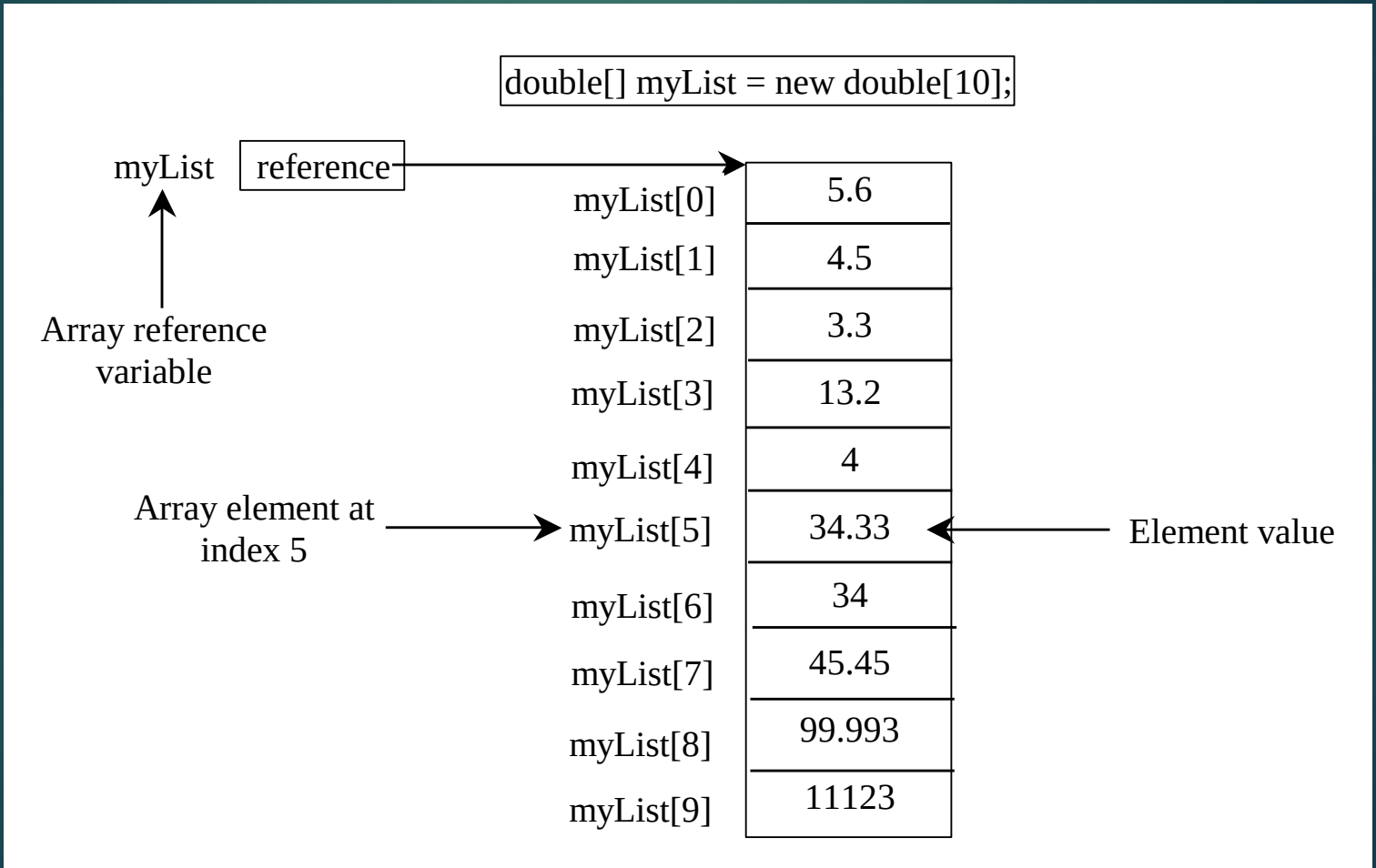
Chapter 6 Arrays

Opening Problem

Read one hundred numbers, compute their average, and find out how many numbers are above the average.

Introducing Arrays

Array is a data structure that represents a collection of the same types of data.



Declaring Array Variables

- `datatype[] arrayRefVar;`

  Example:

  ```
  double[] myList;
  ```

- `datatype arrayRefVar[];` // This style is allowed, but not preferred

  Example:

  ```
  double myList[];
  ```

Creating Arrays


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

Example:

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

`myList[0]` references the first element in the array.

`myList[9]` references the last element in the array.

Declaring and Creating in One Step


- ```
  datatype[] arrayRefVar = new
     datatype[arraySize];
  ```

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

- ```
  datatype arrayRefVar[] = new
    datatype[arraySize];
  ```

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

The Length of an Array

Once an array is created, its size is fixed. It cannot be changed. You can find its size using

arrayRefVar.length

For example,

myList.length returns 10

Default Values

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.

Indexed Variables

The array elements are accessed through the index. The array indices are *0-based*, i.e., it starts from 0 to arrayRefVar.length-1. In the example in Figure 6.1, myList holds ten double values and the indices are from 0 to 9.

Each element in the array is represented using the following syntax, known as an *indexed variable*:

```
arrayRefVar[index];
```

Using Indexed Variables

After an array is created, an indexed variable can be used in the same way as a regular variable. For example, the following code adds the value in myList[0] and myList[1] to myList[2].

```
myList[2] = myList[0] + myList[1];
```

Array Initializers

- Declaring, creating, initializing in one step:

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

This shorthand syntax must be in one statement.

Declaring, creating, initializing, Using the Shorthand Notation


```
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};
```

Initializing arrays with 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();
```

Initializing arrays with random values


```
for (int i = 0; i < myList.length; i++) {
  myList[i] = Math.random() * 100;
}
```

Printing arrays


```
for (int i = 0; i < myList.length; i++) {
  System.out.print(myList[i] + " ");
}
```

Summing all elements


```
double total = 0;
for (int i = 0; i < myList.length; i++) {
  total += myList[i];
}
```

Finding the largest element


```
double max = myList[0];
for (int i = 1; i < myList.length; i++) {
  if (myList[i] > max) max = myList[i];
}
```

Enhanced for Loop (for-each loop)

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 value: myList)
  System.out.println(value);
```

In general, the syntax is

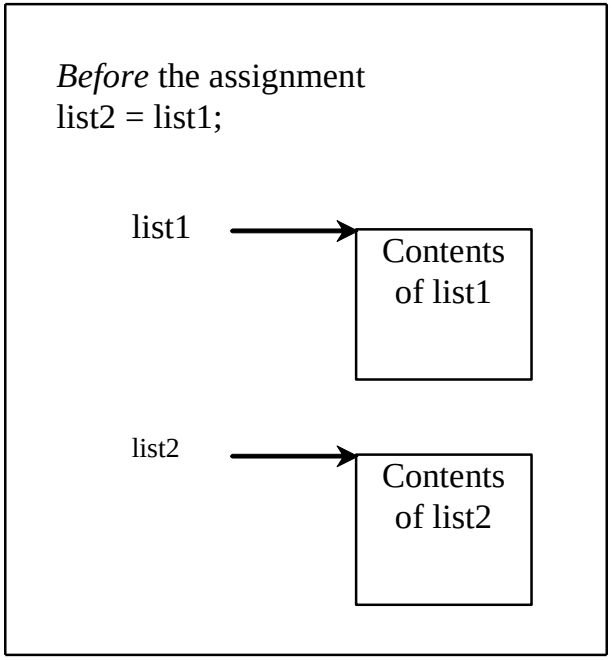
```
for (elementType value: arrayRefVar) {
  // Process the value
}
```

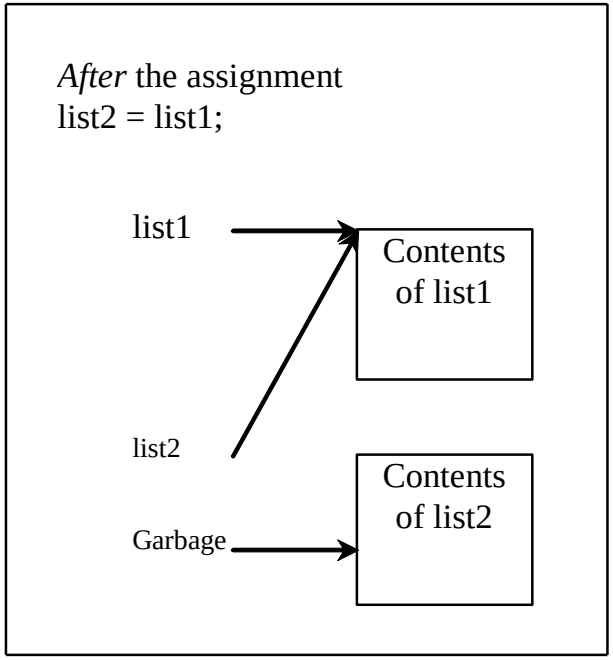
You still have to use an index variable if you wish to traverse the array in a different order or change the elements in the array.

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;
```





Copying 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];
```

Passing Arrays to Methods


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

Invoke the method

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

Invoke the method

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

Anonymous array

Pass By Value

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

- For a parameter of a primitive type value, the actual value is passed. Changing the value of the local parameter inside the method does not affect the value of the variable outside the method.

- For a parameter of an array type, the value of the parameter contains a reference to an array; this reference is passed to the method. Any changes to the array that occur inside the method body will affect the original array that was passed as the argument.

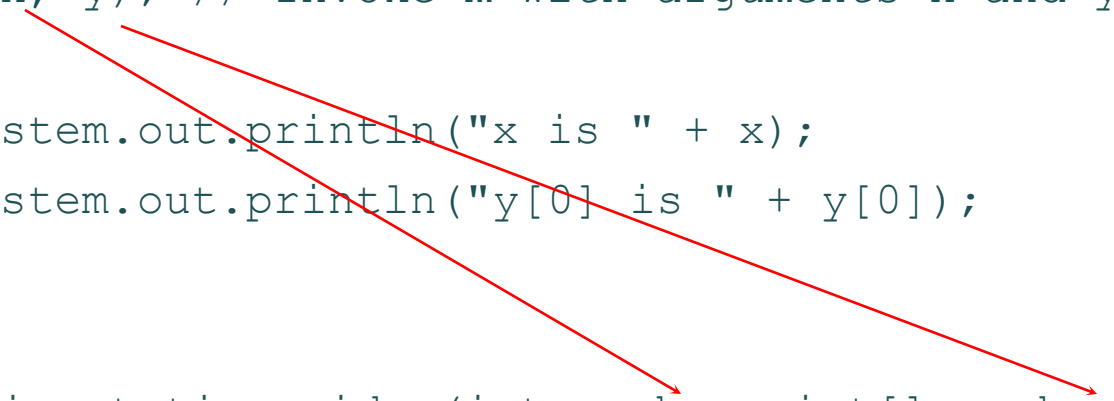
Simple Example


```
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]
  }
}
```



Returning an Array from a Method


```
public static int[] reverse(int[] list) {
  int[] result = new int[list.length];

  for (int i = 0, j = result.length - 1;
       i < list.length; i++, j--) {
    result[j] = list[i];
  }

  return result;
}
```


```
int[] list1 = new int[]{1, 2, 3, 4, 5, 6};
int[] list2 = reverse(list1);
```

The Arrays.binarySearch Method

Since binary search is frequently used in programming, Java provides several overloaded binarySearch methods for searching a key in an array of int, double, char, short, long, and float in the java.util.Arrays class. For example, the following code searches the keys in an array of numbers and an array of characters.

```
int[] list = {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79};
System.out.println("Index is " +
  java.util.Arrays.binarySearch(list, 11));
```

Return is 4

```
char[] chars = {'a', 'c', 'g', 'x', 'y', 'z'};
System.out.println("Index is " +
  java.util.Arrays.binarySearch(chars, 't'));
```

Return is –4 (insertion point is 3, so return is -3-1)

For the binarySearch method to work, the array must be pre-sorted in increasing order.

The Arrays.sort Method

Since sorting is frequently used in programming, Java provides several overloaded sort methods for sorting an array of int, double, char, short, long, and float in the java.util.Arrays class. For example, the following code sorts an array of numbers and an array of characters.

```
double[] numbers = {6.0, 4.4, 1.9, 2.9, 3.4, 3.5};
java.util.Arrays.sort(numbers);

char[] chars = {'a', 'A', '4', 'F', 'D', 'P'};
java.util.Arrays.sort(chars);
```