Chapter 7

Arrays

Goals

This chapter introduces the Java array for storing collections of many objects. Individual elements are referenced with the Java subscript operator []. After studying this chapter you will be able to

- declare and use arrays that can store reference or primitive values
- implement methods that perform array processing

7.1 The Java Array Object

Java **array** objects store collections of elements. They allow a large number of elements to be conveniently maintained together under the same name. The first element is at index 0 and the second is at index 1. Array elements may be any one of the primitive types, such as `int` or `double`. Array elements can also be references to any object.

The following code declares three different arrays named `balance`, `id`, and `tinyBank`. It also initializes all five elements of those three arrays. The subscript operator `[]` provides access to individual array elements.

```
// Declare two arrays that can store up to five elements each
double[] balance = new double[5];
String[] id = new String[5];

// Initialize the array of double values
balance[0] = 0.00;
balance[1] = 111.11;
balance[2] = 222.22;
balance[3] = 333.33;
balance[4] = 444.44;

// Initialize all elements in an array of references to String objects
id[0] = "Bailey";
id[1] = "Dylan";
id[2] = "Hayden";
id[3] = "Madison";
id[4] = "Shannon";
```

The values referenced by the arrays can be drawn like this, indicating that the arrays `balance`, and `id`, store collections. `balance` is a collection of primitive values; `id` is a collection of references to String objects.

balance[0]	0.0
balance[1]	1.11
balance[2]	2.22
balance[3]	3.33
balance[4]	4.44

id[0]	"Bailey"
id[1]	"Dylan"
id[2]	"Hayden"
id[3]	"Madison"
id[4]	"Shannon"

The two arrays above were constructed using the following general forms:

***General Form:* Constructing array objects**

type **[]** *array-name* **= new** *type* **[***capacity***];**

class-name **[]** *array-name* **= new** *class-name* **[***capacity***];**

- *type* specifies the type (either a primitive or reference type) of element that will be stored in the array.
- *array-name* is any valid Java identifier. With subscripts, the array name can refer to any and all elements in the array.
- *capacity* is an integer expression representing the maximum number of elements that can be stored in the array. The capacity is always available through a variable named `length` that is referenced as *array-name*`.length`.

***Example:* array declarations**

```
int[] test = new int[100];            // Store up to 100 integers
double[] number = new double[10000];  // Store up to 10000 numbers
String[] name = new String[500];      // Store up to 500 strings
BankAccount[] customer = new BankAccount[1000]; // 1000 BankAccount references
```

Accessing Individual Elements

Arrays support random access. The individual array elements can be found through subscript notation. A subscript is an integer value between `[` and `]` that represents the index of the element you want to get to. The special symbols `[` and `]` represent the mathematical subscript notation. So instead of x_0, x_1, and x_{n-1}, Java uses `x[0]`, `x[1]`, and `x[n-1]`.

***General Form:* Accessing one array element**

array-name **[***index***]** `// Index should range from 0 to capacity - 1`

The subscript range of a Java array is an integer value in the range of 0 through its capacity - 1. Consider the following array named `x`.

```
double[] x = new double[8];
```

The individual elements of `x` may be referenced using the indexes 0, 1, 2, … 7. If you used `-1` or `8` as an index, you would get an **`ArrayIndexOutOfBoundsException`**. This code assigns values to the first two array elements:

```
// Assign new values to the first two elements of the array named x:
x[0] = 2.6;
x[1] = 5.7;
```

Java uses zero-based indexing. This means that the first array element is accessed with index 0; the same indexing scheme used with `String`. The index 0 means the first element in the collection. With arrays, the first element is found in subscript notation as `x[0]`. The fifth element is accessed with index 4 or with

subscript notation as `x[4]`. This subscript notation allows individual array elements to be displayed, used in expressions, and modified with assignment and input operations. In fact, you can do anything to an individual array element that can be done to a variable of the same type. The array is simply a way to package together a collection of values and treat them as one.

The familiar assignment rules apply to array elements. For example, a `String` literal cannot be assigned to an array element that was declared to store `double` values.

```
// ERROR: x stores numbers, not strings
x[2] = "Wrong type of literal";
```

Since any two `double` values can use the arithmetic operators, numeric array elements can also be used in arithmetic expressions like this:

```
x[2] = x[0] + x[1]; // Store 8.3 into the third array element
```

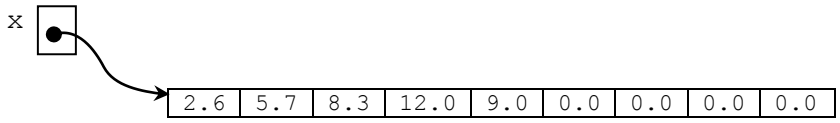
Each array element is a variable of the type declared. Therefore, these two integers will be promoted to double before assignment.

```
x[3] = 12; // Stores 12.0
x[4] = 9;
```

Arrays of primitive `double` values are initialized to a default value of 0.0 (an array of `int`s have elements initialized to 0, arrays of objects to null). The array x originally had all 8 elements to 0.0. After the five assignments above, the array would look like this.

Element Reference	Value
x[0]	2.6
x[1]	5.7
x[2]	8.3
x[3]	12.0
x[4]	9.0
x[5]	0.0
x[6]	0.0
x[7]	0.0

The value of an array is a reference to memory where elements are stored in a contiguous (next to each other) fashion. Here is another view of an array reference value and the elements as the data may exist in the computer's memory.



Out-of-Range Indexes

Java checks array indexes to ensure that they are within the proper range of 0 through capacity - 1. The following assignment results in an exception being thrown. The program usually terminates prematurely with a message like the one shown below.

```
x[8] = 4.5;  // This out-of-range index causes an exception
```

The program terminates prematurely (the output shows the index, which is 8 here).

```
java.lang.ArrayIndexOutOfBounds exception: 8
```

This might seem like a nuisance. However, without range checking, such out-of-range indexes could destroy the state of other objects in memory and cause difficult-to-detect bugs. More dramatically, your computer could "hang" or "crash." Even worse, with a workstation that runs all of the time, you could get an error that affects computer memory now, but won't crash the system until weeks later. However, in Java, you get the more acceptable occurrence of an `ArrayIndexOutOfBounds` exception while you are developing the code.

Self-Check

Use this initialization to answer the questions that follow:

```
int[] arrayOfInts = new int[100];
```

7-1 What type of element can be properly stored as elements in `arrayOfInts`?

7-2 How many integers may be properly stored as elements in `arrayOfInts`?

7-3 Which integer is used as the indearrayOfInts to access the first element in `arrayOfInts`?

7-4 Which integer is used as the indearrayOfInts to access the last element in `arrayOfInts`?

7-5 What is the value of `arrayOfInts[23]`?

7-6 Write code that stores 78 into the first element of `arrayOfInts`.

7-7 What would happen when this code executes? `ArrayOfInts[100] = 100;`

7.2 Array Processing with Determinate Loops

Programmers must frequently access consecutive array elements. For example, you might want to display all of the meaningful elements of an array containing test scores. The Java `for` loop provides a convenient way to do this.

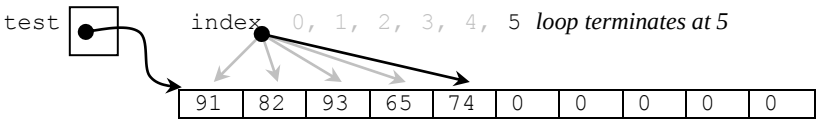
```
int[] test = new int[10];
test[0] = 91;
test[1] = 82;
test[2] = 93;
test[3] = 65;
test[4] = 74;

for (int index = 0; index < 5; index++) {
   System.out.println("test[" + index + "] == " + test[index]);
}
```

Output

```
test[0] == 91
test[1] == 82
test[2] == 93
test[3] == 65
test[4] == 74
```

Changing the `int` variable `index` from 0 through 4 provide accesses to all meaningful elements in the array referenced by `test`. This variable `index` acts both as the loop counter and as an array index inside the `for` loop (`test[index]`). With `index` serving both roles, the specific array element accessed as `test[index]` depends on the value of `index`. For example, when `index` is `0`, `test[index]` references the first element in the array named `test`. When `index` is `4`, `test[index]` references the fifth element. Here is a more graphical view that shows the changing value of index.



Shortcut Array Initialization and the `length` Variable

Java also provides a quick and easy way to initialize arrays without using `new` or the capacity.

```
int[] test = { 91, 82, 93, 65, 74 };
```

The compiler sets the capacity of `test` to be the number of elements between `{` and `}`. The first value (91) is assigned to `test[0]`, the second value (82) to `test[1]`, and so on. Therefore, this shortcut array creation and assignment on one line are equivalent to these six lines of code for a completely filled array (no meaningless values).

```
int[] test = new int[5];
test[0] = 91;
test[1] = 82;
test[2] = 93;
test[3] = 65;
test[4] = 74;
```

This shortcut can be applied to all types.

```
double x[] = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
char[] vowels = { 'a', 'e', 'i', 'o', 'u' };
String[] names = { "Tyler", "Angel", "Justice", "Reese" };
BankAccount[] accounts = {  new BankAccount("Tyler", 100.00),
                            new BankAccount("Angel", 200.00),
                            new BankAccount("Justice", 300.00),
                            new BankAccount("Reese", 400.00)
                         };
```

The `length` variable stores the capacity of an array. It is often used to avoid out-of-range index exceptions. For example, the index range of the array `x` is `0` through `x.length - 1`. The capacity is referenced as the array name, a dot, and the variable named `length`. Do not use `()` after `length` as you would in a `String` message.

```
// Assert the capacities of the four arrays above
assertEquals(7, x.length);
assertEquals(5, vowels.length);
assertEquals(4, names.length);
assertEquals(4, accounts.length);
```

Argument/Parameter Associations

At some point, you will find it necessary to pass an array to another method. In this case, the parameter syntax requires `[]` and the correct type to mark the parameter can be matched to the array argument.

***General Form:* Array parameters**

type **`[]`** *array-reference*

Example Array Parameters in method headings

```
public static void main(String[] args)
public double max(double[] x)
public boolean equal(double[] array1, double[] array2)
```

This allows array references to be passed into a method so that method has access to all elements in the array. For example, this method inspects the meaningful array elements (indexed from 0 through n - 1) to find the smallest value and return it.

```
public int min(int[] array, int n) {
  // Assume the first element is the smallest
  int smallest = array[0];
  // Inspect all other meaningful elements in array[1] through array[n-1]
  for (int index = 1; index < n; index++) {
    if (array[index] < smallest)
      smallest = array[index];
  }
  return smallest;
}
```

An array often stores fewer meaningful elements than its capacity. Therefore, the need arises to store the number of elements in the array that have been given meaningful values. In the previous code, `n` was used to limit the elements being referenced. Only the first five elements were considered to potentially be the smallest. Only the first five should have been considered. Without limiting the search to the meaningful elements (indexed as 0 through n - 1), would the smallest be 65 or would it be one of the 0s stored as one of the fifteen elements at the end that Java initialized to the default value of 0?

Consider the following test method that accidentally passes the array capacity as `test.length` (20) rather than the number of meaningful elements in the array (5).

```
@Test
public void testMin() {
  int[] test = new int[20];
  test[0] = 91;
  test[1] = 82;
  test[2] = 93;
  test[3] = 65;
  test[4] = 74;
  assertEquals(65, min(test, test.length));  // Should be 5
}
```

The assertion fails with this message:

java.lang.AssertionError: expected:<65> but was:<0>

If an array is "filled" with meaningful elements, the length variable can be used to process the array. However, since arrays often have a capacity greater than the number of meaningful elements, it may be better to use some separate integer variable with a name like `n` or `size`.

Messages to Individual Array Elements

The subscript notation must be used to send messages to individual elements. The array name must be accompanied by an index to specify the particular array element to which the message is sent.

***General Form:* Sending messages to individual array elements**

array-name **[***index***].***message-name* **(***arguments)*

The *index* distinguishes the specific object the message is to be sent to. For example, the uppercase equivalent of `id[0]` (this element has the value `"Dylan"`) is returned with this expression:

```
names[0].toUpperCase(); // The first name in an array of Strings
```

The expression `names.toUpperCase()` is a syntax error because it attempts to find the uppercase version of the entire array, not one of its `String` elements. The `toUpperCase` method is not defined for standard Java array objects. On the other hand, `names[0]` does understand `toUpperCase` since `names[0]` is indeed a reference to a `String`. names is a reference to an array of Strings.

Now consider determining the total of all the balances in an array of `BankAccount` objects. The following test method first sets up a miniature database of four `BankAccount` objects. *Note:* A constructor call—with `new`—generates a reference to any type of object. Therefore this assignment

```
// A constructor first constructs an object, then returns its reference
account[0] = new BankAccount("Hall", 50.00);
```

first constructs a `BankAccount` object with the ID `"Hall"` and a balance of `50.0`. The reference to this object is stored in the first array element, `account[0]`.

```
@Test
public void testAssets() {
  BankAccount[] account = new BankAccount[100];
  account[0] = new BankAccount("Hall", 50.00);
  account[1] = new BankAccount("Small", 100.00);
  account[2] = new BankAccount("Ewall", 200.00);
  account[3] = new BankAccount("Westphall", 300.00);
  int n = 4;
  // Only the first n elements of account are meaningful, 96 are null
  double actual = assets(account, n);
  assertEquals(650.00, actual, 0.0001);
}
```

The actual return value from the assets method should be the sum of all account balances indexed from 0..n-1 inclusive, which is expected to be 650.0.

```
// Accumulate the balance of n BankAccount objects stored in account[]
public double assets(BankAccount[] account, int n) {
  double result = 0.0;
  for (int index = 0; index < n; index++) {
    result += account[index].getBalance();
  }
  return result;
}
```

Modifying Array Arguments

Consider the following method that adds the `incValue` to every array element. The test indicates that changes to the parameter `x` also modifies the argument `intArray`.

```
@Test
public void testIncrementBy() {
  int[] intArray = { 1, 5, 12 };
  increment(intArray, 6);
  assertEquals(7, intArray[0]);  // changing the elements of parameter x
  assertEquals(11, intArray[1]); // in increment is the same as changing
  assertEquals(18, intArray[2]); // intArray in this test method
}

public void increment(int[] x, int incValue) {
  for (int index = 0; index < x.length; index++)
    x[index] += incValue;
}
```

To understand why this happens, consider the characteristics of reference variables.

A reference variable stores the location of an object, not the object itself. By analogy, a reference variable is like the address of a friend. It may be a description of where your friend is located, but it is not your actual friend. You may have the addresses of many friends, but these addresses are not your actual friends.

When the Java runtime system constructs an object with the `new` operator, memory for that object gets allocated somewhere in the computer's memory. The `new` operation then returns a reference to that newly constructed object. The reference value gets stored into the reference variable to the left of `=`. For example, the following construction stores the reference to a `BankAccount` object with `"Chris"` and `0.0` into the reference variable named `chris`.

```
BankAccount chris = new BankAccount("Chris", 0.00);
```

A programmer can now send messages to the object by way of the reference value stored in the reference variable named `chris`. The memory that holds the actual state of the object is stored elsewhere. Because you will use the reference variable name for the object, it is intuitive to think of `chris` as the object. However, `chris` is actually the reference to the object, which is located elsewhere.

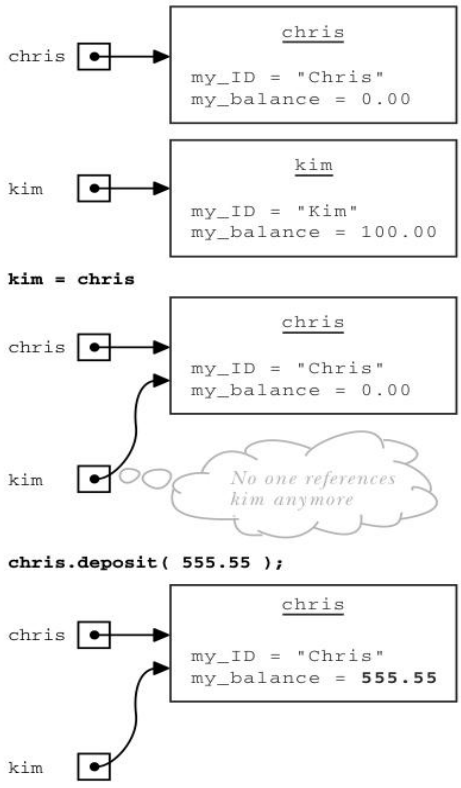
The following code mimics the same assignments that were made to the primitive variables above. The big difference is that the `deposit` message sent to `chris` actually modifies `kim`. This happens because both reference variables `chris` and `kim`—refer to the same object in memory after the assignment `kim = chris`. In fact, the object originally referred to by the reference variable named `kim` is lost forever. Once the memory used to store the state of an object no longer has any references, Java's garbage collector reclaims the memory so it can be reused later to store other new objects. This allows your computer to recycle memory that is no longer needed.

```
BankAccount chris = new BankAccount("Chris", 0.00);
BankAccount kim = new BankAccount("Kim", 100.00);
kim = chris;
// The values of the object were not assigned.
// Rather, the reference to chris was assigned to the reference variable kim.
// Now both reference variables refer to the same object.
System.out.println("Why does a change to 'chris' change 'kim'?");
chris.deposit(555.55);
System.out.println("Kim's balance was 0.00, now it is " + kim.getBalance());
```

Output

```
Why does a change to 'chris' change 'kim'?
Kim's balance was 0.00, now it is 555.55
```

Assignment statements copy the values to the right of = into the variable to the left of =. When the variables are primitive number types like `int` and `double`, the copied values are numbers. However, when the variables are references to objects, the copied values are the locations of the objects in memory as illustrated in the following diagram.



After the assignment `kim = chris`, `kim` and `chris` both refer to the same object in memory. The state of the object is not assigned. Intead, the reference to the object is assigned. A message to either reference variable (`chris` or `kim`) accesses or modifies the same object, which now has the state of "Chris" and 555.55. An assignment of a reference value to another reference variable of the same type does not change the object itself. The state of an object can only be changed with messages designed to modify the state.

The big difference is that the `deposit` message to `chris` actually modified `kim`. This happens because both reference variables—`chris` and `kim`—refer to the same object in memory after the assignment `kim = chris`.

The same assignment rules apply when an argument is assigned to a parameter. In this method and test, chris and kim both refer to the same object.

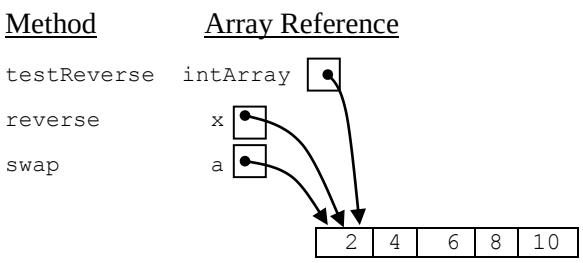
```
@Test
public void testAddToBalance() {
  BankAccount kim = new BankAccount("Chris", 0.00);
  assertEquals(0.0, kim.getBalance(), 0.0001);
  increment(kim);
  assertEquals(555.55, kim.getBalance(), 1e-14);
}

public void increment(BankAccount chris) {
  chris.deposit(555.55);
}
```

Java has one argument/parameter association. It is called pass by value. When an argument is assigned to a parameter, the argument's value is copied to the parameter. When the argument is a primitive type such as `int` or `double`, the copied values are primitive numeric values or `char` values. No method can change the primitive arguments of another method. However, when an object reference is passed to a method, the value is a reference value. The argument is the location of the object in computer memory.

At that moment, the parameter is an alias (another name) for the argument. Two references to the same object exist. The parameter refers to the same object as the argument. This means that when a method modifies the parameter, the change occurs in the object referenced by the argument.

In this code that reverses the array elements, three reference variables reference the array of ints constructed in the test method.




```
@Test
public void testReverse() {
  int[] intArray = { 2, 4, 6, 8, 10 };
  reverse(intArray);
  assertEquals(10, intArray[0]); // was 2
  assertEquals(8, intArray[1]);  // was 4
  assertEquals(6, intArray[2]);  // was 6
  assertEquals(4, intArray[3]);  // was 8
  assertEquals(2, intArray[4]);  // was 10
}
```

```
// Reverse the array elements so x[0] gets exchanged with x[x.length-1],
// x[1] with x[x.length-2], x[2] with x[x.length-3], and so on.
public void reverse(int[] x) {
  int leftIndex = 0;
  int rightIndex = x.length - 1;
  while (leftIndex < rightIndex) {
    swap(x, leftIndex, rightIndex);
    leftIndex++;
    rightIndex--;
  }
}

// Exchange the two integers in the specified indexes
// inside the array referenced by a.
private void swap(int[] a, int leftIndex, int rightIndex) {
  int temp = a[leftIndex];      // Need to store a[leftIndex] before
  a[leftIndex] = a[rightIndex]; // a[leftIndex] gets erased in this assignment
  a[rightIndex] = temp;
}
```

Self-Check

7-8 Given the small change of < to <= in the for loop, describe what would happen when this method is called where the number of meaningful elements is n.

```
// Accumulate the balance of n BankAccount objects stored in account[]
public double assets(BankAccount[] account, int n) {
  double result = 0.0;
  for (int index = 0; index <= n; index++) {
    result += account[index].getBalance();
  }
  return result;
}
```

7-9 Write method `sameEnds` to return true if the integer in the first index equals the integer in the last index. This code must compile and the assertions must pass.

```
@Test public void testSameEnds() {
  int[] x1 = { 1, 2, 3, 4, 5 };
  int[] x2 = { 4, 3, 2, 1, 0, 1, 2, 4 };
  int[] x3 = { 5, 6 };
  int[] x4 = { 5, 5 };
  assertFalse(sameEnds(x1));
  assertTrue(sameEnds(x2));
  assertFalse(sameEnds(x3));
  assertTrue(sameEnds(x4));
}
```

7-10 Write method `swapEnds` that switches the end elements in an array of Strings. The following code must compile and the assertions must pass.

```
@Test public void testSwapEnds() {
  String[] strings = { "a", "b", "c", "x" };
  swapEnds(strings);
  assertEquals("x", strings[0]);
  assertEquals("b", strings[1]);
  assertEquals("c", strings[2]);
  assertEquals("a", strings[3]);
}

@Test public void testSwapEndsWhenLengthIsTwo() {
  String[] strings = { "a", "x" };
  swapEnds(strings);
  assertEquals("x", strings[0]);
  assertEquals("a", strings[1]);
}
```

```
@Test public void testSwapEndsWhenTooSmall() {
  String[] strings = { "a" };
  // There should be no exceptions thrown. Use guarded action.
  swapEnds(strings);
  assertEquals("a", strings[0]);
}
```

7-11 Write method for `accountsLargerThan` that takes an array of BankAccount references s and returns the number of accounts with a balance greater than the second argument of type double. The following test method must compile and the assertions must pass.

```
@Test
public void testAssets() {
  BankAccount[] account = new BankAccount[100];
  account[0] = new BankAccount("Hall", 50.00);
  account[1] = new BankAccount("Small", 100.00);
  account[2] = new BankAccount("Ewall", 200.00);
  account[3] = new BankAccount("Westphall", 300.00);
  int n = 4;

  int actual = studentsFun.accountsLargerThan(0.00, account, n);
  assertEquals(4, actual);
  actual = studentsFun.accountsLargerThan(50.00, account, n);
  assertEquals(3, actual);
  actual = studentsFun.accountsLargerThan(100.00, account, n);
  assertEquals(2, actual);
  actual = studentsFun.accountsLargerThan(200.00, account, n);
  assertEquals(1, actual);
  actual = studentsFun.accountsLargerThan(300.00, account, n);
  assertEquals(0, actual);
}
```

Answers to Self-Checks

7-1 `int` 7-2 100 7-3 0 7-4 99 7-5 0 7-6 `x[0] = 78;`

7-7 `ArrayIndexOutOfBounds` exception would terminate the program

7-8 There would be a `getBalance()` message sent to account[n+1] which is probably null. Program terminates

7-9
```
public boolean sameEnds(int[] array) {
    return array[0] == array[array.length-1];
  }
```

7-10
```
private void swapEnds(String[] array) {
    if (array.length >= 2) {
      int rightIndex = array.length - 1;
      String temp = array[rightIndex];
      array[rightIndex] = array[0];
      array[0] = temp;
    }
  }
```

7-11
```
public int accountsLargerThan(double amt, BankAccount[] account, int n) {
    int result = 0;
    for (int index = 0; index < n; index++) {
      if(account[index].getBalance() > amt)
        result ++;
    }
    return result;
  }
```