GLOBAL EDITION

Absolute JAVA

SIXTH EDITION

Walter Savitch

ALWAYS LEARNING

PEARSON

ABSOLUTE JAVA™

6th Edition
Global Edition

6.2 Arrays and References

A little more than kin, and less than kind.

WILLIAM SHAKESPEARE, *Hamlet, 1603.*

Just like a variable of one of the class types you have seen, a variable of an array type holds a reference. In this section, we explore the consequences of this fact, including a discussion of array parameters. We will see that arrays are objects and that array types can be considered class types but somewhat different kinds of class types than what you are used to. Arrays and the kinds of classes we have seen before this chapter are *a little more than kin, and less than kind.*

Arrays Are Objects

There are two ways to view an array: as a collection of indexed variables and as a single item whose value is a collection of values of the base type. In Section 6.1, we discussed using arrays as a collection of indexed variables. We will now discuss arrays from the second point of view.

An array can be viewed as a single item whose value is a collection of values of the base type. An array variable (as opposed to an array indexed variable) names the array as a single item. For example, the following declares a variable of an array type:

```
double[] a;
```

This variable `a` can and will contain a single value. The expression

```
new double[10]
```

creates an array object and stores the object in memory. The following assignment statement places a reference to (the memory address of) this array object in the variable `a`:

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

Typically, we combine all this into a single statement as follows:

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

Notice that this is almost exactly the same as the way that we view objects of a class type. In Java, an array is considered an object. Whenever Java documentation says that something applies to objects, it means that it applies to arrays as well as objects of the class types we have seen up to now. You will eventually see examples of methods that can take arguments that may be objects of any kind. These methods will accept array objects as arguments as well as objects of an ordinary class type. Arrays are somewhat peculiar in how they relate to classes. Some authorities say array types are not classes, and some say they are. But, all authorities agree that the arrays themselves are objects. Given that arrays are objects, it seems that one should view array types as classes, and we will do so. However, although an array type `double[]` is a class, the syntax for creating an array object is a bit different. To create an array, use the following syntax:

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

You can view the expression `new double[10]` as an invocation of a constructor that uses a nonstandard syntax. (The nonstandard syntax was used to be consistent with the syntax used for arrays in older programming languages.)

As we have already seen, every array has an instance variable named `length`, which is a good example of viewing an array as an object. As with any other class type, array variables contain memory addresses, or, as they are usually called in Java, *references*. So, array types are reference types.[3]

Since an array is an object, you might be tempted to think of the indexed variables of an array, such as `a[0]`, `a[1]`, and so forth, as being instance variables of the object. This is actually a pretty good analogy, but it is not literally true. Indexed variables are not instance variables of the array. Indexed variables are a special kind of variable peculiar to arrays. The only instance variable in an array is the `length` instance variable.

An array object is a collection of items of the base type. Viewed as such, an array is an object that can be assigned with the assignment operator and plugged in for a parameter of an array type. Because an array type is a reference type, the behaviors of arrays with respect to assignment `=`, `==`, and parameter passing mechanisms are the same as what we have already described for classes. In the next few subsections, we discuss these details about arrays.

Arrays Are Objects

In Java, arrays are considered to be objects, and, although there is some disagreement on this point, you can safely view an array type as a class type.

Array Types Reference Types

A variable of an array type holds the address of where the array object is stored in memory. This memory address is called a **reference** to the array object.

[3]In many programming languages, such as C++, arrays are also reference types just as they are in Java. So, this detail about arrays is not peculiar to Java.

PITFALL: Arrays with a Class Base Type

The base type of an array can be of any type, including a class type. For example, suppose `Date` is a class and consider the following:

```
Date[] holidayList = new Date[20];
```

VideoNote
Arrays of Objects

This creates the 20 indexed variables (`holidayList[0]`, `holidayList[1]`, ..., `holidayList[19]`). It is important to note that this creates 20 *indexed variables* of type `Date`. This does not create 20 *objects* of type `Date`. (The index variables are automatically initialized to `null`, not to an object of the class `Date`.) Like any other variable of type `Date`, the indexed variables require an invocation of a constructor using `new` to create an object. One way to complete the initialization of the array `holidayList` is as follows:

```
Date[] holidayList = new Date[20];
for (int i = 0; i < holidayList.length; i++)
    holidayList[i] = new Date( );
```

If you omit the `for` loop (and do not do something else more or less equivalent), then when you run your code, you will undoubtedly get an error message indicating a "null pointer exception." If you do not use `new` to create an object, an indexed variable like `holidayList[i]` is just a variable that names no object and hence cannot be used as the calling object for any method. Whenever you are using an array with a class base type and you get an error message referring to a "Null Pointer Exception," it is likely that your indexed variables do not name any objects and you need to add something such as the above `for` loop. ■

Array Parameters

You can use both array indexed variables and entire arrays as arguments to methods, although they are different types of parameters. We first discuss array indexed variables as arguments to methods.

indexed variable arguments

An indexed variable can be an argument to a method in exactly the same way that any variable of the array base type can be an argument. For example, suppose a program contains the following declarations:

```
double n = 0;
double[] a = new double[10];
int i;
```

If `myMethod` takes one argument of type `double`, then the following is legal:

```
myMethod(n);
```

Since an indexed variable of the array a is also a variable of type double, just like n, the following is equally legal:

```
myMethod(a[3]);
```

There is one subtlety that does apply to indexed variables used as arguments. For example, consider the following method call:

```
myMethod(a[i]);
```

If the value of i is 3, then the argument is a[3]. On the other hand, if the value of i is 0, then this call is equivalent to the following:

```
myMethod(a[0]);
```

The indexed expression is evaluated to determine exactly which indexed variable is given as the argument.

Array Indexed Variables as Arguments

An array indexed variable can be used as an argument anyplace that a variable of the array's base type can be used. For example, suppose you have the following:

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

Indexed variables such as a[3] and a[index] can then be used as arguments to any method that accepts a double as an argument.

entire array parameters

You can also define a method that has a formal parameter for an entire array so that when the method is called, the argument that is plugged in for this formal parameter is an entire array. Whenever you need to specify an array type, the type name has the form *Base_Type*[], so this is how you specify a parameter type for an entire array. For example, the method doubleArrayElements, given in what follows, will accept any array of double as its single argument:

```
public class SampleClass
{
    public static void doubleArrayElements(double[] a)
    {
        int i;
        for (i = 0; i < a.length; i++)
           a[i] = a[i]*2;
    }
    < The rest of the class definition goes here.>
}
```

To illustrate this, suppose you have the following in some method definition:

```
double[] a = new double[10];
double[] b = new double[30];
```

and suppose that the elements of the arrays a and b have been given values. Both of the following are then legal method invocations:

```
SampleClass.doubleArrayElements(a);
SampleClass.doubleArrayElements(b);
```

Note that no square brackets are used when you give an entire array as an argument to a method.

An array type is a reference type just as a class type is, so, as with a class type argument, a method can change the data in an array argument. To phrase it more precisely, a method can change the values stored in the indexed variables of an array argument. This is illustrated by the preceding method `doubleArrayElements`.

length of array arguments

An array type parameter does not specify the length of the array argument that may be plugged in for the parameter. An array knows its length and stores it in the `length` instance variable. The same array parameter can be replaced with array arguments of different lengths. Note that the preceding method `doubleArrayElements` can take an array of any length as an argument.

PITFALL: Use of = and == with Arrays

assignment with arrays

Array types are reference types; that is, an array variable contains the memory address of the array it names. The assignment operator copies this memory address. For example, consider the following code:

```
double[] a = new double[10];
double[] b = new double[10];
int i;
for (i = 0; i < a.length; i++)
    a[i] = i;
b = a;
System.out.println("a[2] = " + a[2] + " b[2] = " + b[2]);
a[2] = 42;
System.out.println("a[2] = " + a[2] + " b[2] = " + b[2]);
```

This will produce the following output:

```
a[2] = 2.0 b[2] = 2.0
a[2] = 42.0 b[2] = 42.0
```

PITFALL: (continued)

The assignment statement `b = a;` copies the memory address from `a` to `b` so that the array variable `b` contains the same memory address as the array variable `a`. After the assignment statement, `a` and `b` are two different names for the same array. Thus, when we change the value of `a[2]`, we are also changing the value of `b[2]`.

Unless you want two array variables to be two names for the same array (and on rare occasions, you do want this), you should not use the assignment operator with arrays. If you want the arrays `a` and `b` in the preceding code to be different arrays with the same values in each index position, then instead of the assignment statement

```
b = a;
```

you need to use something such as the following:

```
int i;
for (i = 0; (i < a.length) && (i < b.length); i++)
    b[i] = a[i];
```

Note that the above code will not make `b` an exact copy of `a`, unless `a` and `b` have the same length.

== with arrays

The equality operator `==` does not test two arrays to see if they contain the same values. It tests two arrays to see if they are stored in the same location in the computer's memory. For example, consider the following code:

```
int[] c = new int[10];
int[] d = new int[10];
int i;
for (i = 0; i < c.length; i++)
    c[i] = i;
for (i = 0; i < d.length; i++)
    d[i] = i;

if (c == d)
    System.out.println("c and d are equal by ==.");
else
    System.out.println("c and d are not equal by ==.");
```

This produces the output

```
c and d are not equal by ==
```

(continued)