[Next Section | Chapter Index | Main Index]

Section 7.1

Array Details

Subsections

ARRAY BASICS HAVE BEEN DISCUSSED in previous chapters, but there are still details of Java syntax to be filled in, and there is a lot more to say about using arrays. This section looks at some of the syntactic details, with more information about array processing to come in the rest of the chapter.

To briefly review some of the basics.... An array is a numbered sequence of *elements*, and each element acts like a separate variable. All of the elements are of the same type, which is called the *base type* of the array. The array as a whole also has a type. If the base type is btype, then the array is of type `btype[]`. Each element in the array has an *index*, which is just its numerical position in the sequence of elements. If the array is `A`, then the i-th element of the array is `A[i]`. The number of elements in an array is called its *length*. The length of an array `A` is `A.length`. The length of an array can't be changed after the array is created. The elements of the array `A` are `A[0]`, `A[1]`, ..., `A[A.length-1]`. An attempt to refer to an array element with an index outside the range from zero to `A.length-1` causes an ArrayIndexOutOfBoundsException.

Arrays in Java are objects, so an array variable can only refer to an array, it does not contain the array. The value of an array variable can also be `null`. In that case, it does not refer to any array, and an attempt to refer to an array element such as `A[i]` will cause a NullPointerException. Arrays are created using a special form of the `new` operator. For example,

```
int[] A = new int[10];
```

creates a new array with base type int and length 10, and it sets the variable `A` to refer to the newly created array.

7.1.1 For-each Loops

Arrays are often processed using `for` loops. A `for` loop makes it easy to process each element in an array from beginning to end. For example, if `namelist` is an array of Strings, then all the values in the list can be printed using

```
for (int i = 0; i < namelist.length; i++) {
    System.out.println( namelist[i] );
}
```

This type of processing is so common that there is an alternative form of the `for` loop that makes it easier to write. The alternative is called a for-each loop. It is probably easiest to start with an example. Here is a for-each loop for printing all the values in an array of Strings:

```
for ( String name : namelist ) {
    System.out.println( name );
}
```

The meaning of "`for (String name : namelist)`" is "for each string, name, in the array, namelist, do the following". The effect is that the variable `name` takes on each of the values in `namelist` in turn, and the body of the loop is executed for each of those values. Note that there is no array index in the loop.

The loop control variable, `name`, represents one of the values in the array, not the index of one of the values.

The for-each loop is meant specifically for processing all the values in a data structure, and we will see in Chapter 10 that it applies to other data structures besides arrays. The for-each loop makes it possible to process the values without even knowing the details of how the data is structured. In the case of arrays, it lets you avoid the complications of using array indices.

A for-each loop will perform the same operation for each value that is stored in an array. If `itemArray` is an array of type `BaseType[]`, then a for-each loop for `itemArray` has the form:

```
for ( BaseType item : itemArray ) {
   .
   .  // process the item
   .
}
```

As usual, the braces are optional if there is only one statement inside the loop. In this loop, `item` is the loop control variable. It is declared as a variable of type BaseType, where BaseType is the base type of the array. (In a for-each loop, the loop control variable **must** be declared in the loop; it cannot be a variable that already exists outside the loop.) When this loop is executed, each value from the array is assigned to `item` in turn and the body of the loop is executed for each value. Thus, the above loop is exactly equivalent to:

```
for ( int index = 0; index < itemArray.length; index++ ) {
   BaseType item;
   item = itemArray[index];  // Get one of the values from the array
     .
     .  // process the item
     .
}
```

For example, if `A` is an array of type `int[]`, then we could print all the values from `A` with the for-each loop:

```
for ( int item : A )
   System.out.println( item );
```

and we could add up all the positive integers in `A` with:

```
int sum = 0;   // This will be the sum of all the positive numbers in A
for ( int item : A ) {
   if (item > 0)
      sum = sum + item;
}
```

The for-each loop is not always appropriate. For example, there is no simple way to use it to process the items in just a part of an array, or to process the elements in reverse order. However, it does make the code a little simpler when you do want to process all the values, in order. since it eliminates any need to use array indices.

It's important to note that a for-each loop processes the **values** in the array, not the **elements** (where an element means the actual memory location that is part of the array). For example, consider the following incorrect attempt to fill an array of integers with 17's:

```
int[] intList = new int[10];
for ( int item : intList ) {   // INCORRECT! DOES NOT MODIFY THE ARRAY!
   item = 17;
}
```

The assignment statement `item = 17` assigns the value 17 to the loop control variable, `item`. However, this has nothing to do with the array. When the body of the loop is executed, the value from one of the elements of the array is copied into `item`. The statement `item = 17` replaces that copied value but has no effect on the array element from which it was copied; the value in the array is not changed. The loop is equivalent to

```
int[] intList = new int[10];
for ( int i = 0; i < intList.length; i++ ) {
   int item = intList[i];
   item = 17;
}
```

which certainly does not change the value of any element in the array.

7.1.2 Variable Arity Methods

Before Java 5, every method in Java had a fixed arity. (The arity of a method is defined as the number of parameters in a call to the method.) In a fixed arity method, the number of parameters must be the same in every call to the method and must be the same as the number of formal parameters in the method's definition. Java 5 introduced variable arity methods. In a variable arity method, different calls to the method can have different numbers of parameters. For example, the formatted output method `System.out.printf`, which was introduced in Subsection 2.4.1, is a variable arity method. The first parameter of `System.out.printf` must be a String, but it can have any number of additional parameters, of any types.

Calling a variable arity method is no different from calling any other sort of method, but writing one requires some new syntax. As an example, consider a method that can compute the average of any number of values of type double. The definition of such a method could begin with:

```
public static double average( double...  numbers ) {
```

Here, the `...` after the type name, `double`, is what makes this a variable arity method. It indicates that any number of values of type double can be provided when the subroutine is called, so that for example `average(1,4,9,16)`, `average(3.14,2.17)`, `average(0.375)`, and even `average()` are all legal calls to this method. Note that actual parameters of type int can be passed to `average`. The integers will, as usual, be automatically converted to real numbers.

When the method is called, the values of all the actual parameters that correspond to the variable arity parameter are placed into an array, and it is this array that is actually passed to the method. That is, in the body of a method, a variable arity parameter of type T actually looks like an ordinary parameter of type `T[]`. The length of the array tells you how many actual parameters were provided in the method call. In the `average` example, the body of the method would see an array named `numbers` of type `double[]`. The number of actual parameters in the method call would be `numbers.length`, and the values of the actual parameters would be `numbers[0]`, `numbers[1]`, and so on. A complete definition of the method would be:

```
public static double average( double... numbers ) {
        // Inside this method, numbers if of type double[].
   double sum;      // The sum of all the actual parameters.
   double average;  // The average of all the actual parameters.
   sum = 0;
   for (int i = 0; i < numbers.length; i++) {
      sum = sum + numbers[i];  // Add one of the actual parameters to the sum.
   }
   average = sum / numbers.length;
```

```
   return average;
}
```

By the way, it is possible to pass a single array to a variable arity method, instead of a list of individual values. For example, suppose that `salesData` is a variable of type `double[]`. Then it would be legal to call `average(salesData)`, and this would compute the average of all the numbers in the array.

The formal parameter list in the definition of a variable-arity method can include more than one parameter, but the `...` can only be applied to the very last formal parameter.

As an example, consider a method that can draw a polygon through any number of points. The points are given as values of type Point, where an object of type Point has two instance variables, `x` and `y`, of type int. In this case, the method has one ordinary parameter -- the graphics context that will be used to draw the polygon -- in addition to the variable arity parameter. Remember that inside the definition of the method, the parameter `points` becomes an array of Points:

```
public static void drawPolygon(Graphics g, Point... points) {
    if (points.length > 1) {  // (Need at least 2 points to draw anything.)
       for (int i = 0; i < points.length - 1; i++) {
           // Draw a line from i-th point to (i+1)-th point
           g.drawLine( points[i].x, points[i].y, points[i+1].x, points[i+1].y );
       }
       // Now, draw a line back to the starting point.
       g.drawLine( points[points.length-1].x, points[points.length-1].y,
                      points[0].x, points[0].y );
    }
}
```

When this method is called, the subroutine call statement must have one actual parameter of type Graphics, which can be followed by any number of actual parameters of type Point.

For a final example, let's look at a method that strings together all of the values in a list of strings into a single, long string. This example uses a for-each loop to process the array:

```
public static String concat( String... values ) {
   StringBuffer buffer;  // Use a StringBuffer for more efficient concatenation.
   buffer = new StringBuffer();  // Start with an empty buffer.
   for ( String str : values ) { // A "for each" loop for processing the values.
       buffer.append(str); // Add string to the buffer.
   }
   return buffer.toString(); // return the contents of the buffer
}
```

Given this method definition, `concat("Hello", "World")` would return the string "HelloWorld", and `concat()` would return an empty string. Since a variable arity method can also accept an array as actual parameter, we could also call `concat(lines)` where `lines` is of type `String[]`. This would concatenate all the elements of the array into a single string.

7.1.3 Array Literals

We have seen that it is possible to initialize an array variable with a list of values at the time it is declared. For example,

```
int[] squares = { 1, 4, 9, 16, 25, 36, 49 };
```

This initializes `squares` to refer to a newly created array that contains the seven values in the list

A list initializer of this form can be used **only** in a declaration statement, to give an initial value to a newly declared array variable. It cannot be used in an assignment statement to assign a value to a variable that already existed. However, there is another, similar notation for creating a new array that can be used in other places. The notation uses another form of the `new` operator to both create a new array object and fill it with values. (The rather odd syntax is similar to the syntax for anonymous inner classes, which were discussed in Subsection 5.8.3.) As an example, to assign a new value to an array variable, `cubes`, of type `int[]`, you could use:

```
cubes = new int[] { 1, 8, 27, 64, 125, 216, 343 };
```

This is an assignment statement rather than a declaration, so the array initializer syntax, without "`new int[]`," would not be legal here. The general syntax for this form of the `new` operator is

```
new base-type [ ] { list-of-values }
```

This is actually an expression whose value is a reference to a newly created array object. In this sense, it is an "array literal," since it is something that you can type in a program to represent a value. This means that it can be used in any context where an object of type **base-type**`[]` is legal. For example, you could pass the newly created array as an actual parameter to a subroutine. Consider the following utility method for creating a menu from an array of strings:

```
/**
 * Creates a JMenu.  The names for the JMenuItems in the menu are
 * given as an array of strings.
 * @param menuName  the name for the JMenu that is to be created.
 * @param handler  the listener that will respond to items in the menu.
 *    This ActionListener is added as a listener for each JMenuItem.
 * @param itemNames  an array holding the text that appears in each
 *     JMenuItem.  If a null value appears in the array, the corresponding
 *     item in the menu will be a separator rather than a JMenuItem.
 * @return  the menu that has been created and filled with items.
 */
public static JMenu createMenu(
            String menuName, ActionListener handler, String[] itemNames ) {
    JMenu menu = new JMenu(menuName);
    for ( String itemName : itemNames ) {
        if ( itemName == null ) {
            menu.addSeparator();
        }
        else {
            JMenuItem item = new JMenuItem(itemName);
            item.addActionListener(handler);
            menu.add(item);
        }
    }
    return menu;
}
```

The third parameter in a call to `createMenu` is an array of strings. The array that is passed as an actual parameter could be created in place, using the `new` operator. For example, assuming that `listener` is of type ActionListener, we can use the following statement to create an entire File menu:

```
JMenu fileMenu = createMenu( "File", listener
              new String[] { "New", "Open", "Close", null, "Quit" } );
```

This should convince you that being able to create and use an array "in place" in this way can be very convenient, in the same way that anonymous inner classes are convenient.

By the way, it is perfectly legal to use the "`new BaseType[] { ... }`" syntax instead of the array initializer syntax in the declaration of an array variable. For example, instead of saying:

```
int[] primes = { 2, 3, 5, 7, 11, 13, 17, 19 };
```

you can say, equivalently,

```
int[] primes = new int[] { 2, 3, 5, 7, 11, 17, 19 };
```

In fact, rather than use a special notation that works only in the context of declaration statements, I sometimes prefer to use the second form.

One final note: For historical reasons, an array declaration such as

```
int[] list;
```

can also be written as

```
int list[];
```

which is a syntax used in the languages C and C++. However, this alternative syntax does not really make much sense in the context of Java, and it is probably best avoided. After all, the intent is to declare a variable of a certain type, and the name of that type is "`int[]`". It makes sense to follow the "**type-name variable-name**;" syntax for such declarations.

[Next Section | Chapter Index | Main Index]