ARRAYS, FOR-EACH LOOPS AND ARRAY TYPES IN SUBROUTINES

Arrays and for-each Loops :

Java 5.0 introduced a new form of the `for` loop, the "for-each loop" that was discussed in Subsection 3.4.4. The for-each loop is meant specifically for processing all the values in a data structure. When used to process an array, a for-each loop can be used to perform the same operation on each value that is stored in the array.
If `anArray` is an array of type `BaseType[]`, then a for-each loop
for `anArray` has the form:

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

In this loop, `item` is the loop control variable. It is being 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.) 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 < anArray.length; index++ ) {
   BaseType item;
   item = anArray[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. However, it does make it a little easier to process all the values in an array, 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.

Array Types in Subroutines

Any array type, such as `double[]`, is a full-fledged Java type, so it can be used in all the ways that any other Java type can be used. In particular, it can be used as the type of a formal parameter in a subroutine. It can even be the return type of a function. For example, it might be useful to have a function that makes a copy of an array of `double`:

```
/**
 *  Create a new array of doubles that is a copy of a given
array.
 *  @param source the array that is to be copied; the value can
be null
 *  @return a copy of source; if source is null, then the
return value is also null
 */
public static double[]  copy( double[] source ) {
    if ( source == null )
```

```
        return null;
    double[]  cpy;  // A copy of the source array.
    cpy = new double[source.length];
    System.arraycopy( source, 0, cpy, 0, source.length );
    return cpy;
}
```

The `main()` routine of a program has a parameter of type `String[]`. You've seen this used since all the way back in Section 2.1, but I haven't really been able to explain it until now. The parameter to the`main()` routine is an array of *Strings*. When the system calls the `main()` routine, it passes an actual array of strings, which becomes the value of this parameter. Where do the strings come from? The strings in the array are the command-line arguments from the command that was used to run the program. When using a command-line interface, the user types a command to tell the system to execute a program. The user can include extra input in this command, beyond the name of the program. This extra input becomes the command-line arguments. For example, if the name of the class that contains the `main()` routine is`myProg`, then the user can type "`java myProg`" to execute the program. In this case, there are no command-line arguments. But if the user types the command

```
java myProg one two three
```

then the command-line arguments are the strings "one", "two", and "three". The system puts these strings into an array of *Strings* and passes that array as a parameter to the `main()` routine. Here, for example, is a short program that simply prints out any command line arguments entered by the user:

```
public class CLDemo {

   public static void main(String[] args) {
      System.out.println("You entered " + args.length
                                    + " command-line arguments");
      if (args.length > 0) {
         System.out.println("They were:");
         for (int i = 0; i < args.length; i++)
            System.out.println("   " + args[i]);
      }
   } // end main()

} // end class CLDemo
```

Note that the parameter, `args`, is never `null` when `main()` is called by the system, but it might be an array of length zero.

In practice, command-line arguments are often the names of files to be processed by the program. I will give some examples of this in Chapter 11, when I discuss file processing.

Source : http://math.hws.edu/javanotes/c7/s2.html