Lesson 21: Advanced Array Concepts

Arrays of objects:

```
Circle cir[] = new Circle[500];        //declares 500 circles, all null for the moment

//We can initialize each of these 500 Circle objects individually as shown here
cir[117] = new Circle(57.2);           //set radius to 57.2

for (int j = 0; j < 500; j++)          //…or we can initialize them in a loop
{
        cir[j] = new Circle(10);       //all radii set to 10
}
```

Comparison of array values:

We will give examples of *boolean* values within fragments of ***if*** statements; however, any other such usage of *boolean* values using arrays would be acceptable:

a. **Numeric** arrays:

```
if ( n[23] = = n[k+1] )
if ( n[23] >= n[k+1] )
```

b. **String** arrays:

```
if ( s[3 +d] .equals("hermit") )
if ( s[3 +d] .compareTo("hermit") > 0 )
```

c. **Object** arrays:

```
if ( BankAccount[1].equals(BankAccount[2]) )
```

The dreaded *NullPointerException*:

```
double mxz[];          //the array mxz has only been declared
mxz[3] = 19.1;         //error! NullPointerException, mxz has not been initialized yet.
```

Different references to the same array:

Because arrays are objects, two or more variables can refer to the same array as in the following example:

```
int []frst = {1, 2, 3, 4, 5};      // frst[] declared and initialized
int sec[];                         // sec[] is just declared

sec = frst;
sec[2] = 99;
System.out.println(frst[2]);       //99 Notice that even though we changed only
                                   //sec[2] to 99, frst[2] also changes to 99.
```

Declaring multiple arrays…which to use, []x or x[]?

When declaring multiple arrays on a single line, the placement of [] is critical.

```
int[] x, y; //Both x and y are arrays.
int x[], y; //Only x is an array.
```

Removing an array from memory:

It is possible for the *GarbageCollector* to release the memory of an array (or any object). To enable this, simply set **all references** to the array (or object) equal to **null** as follows:

```
int myArray[] = new int[500]; //occupies 500 * 4 bytes of memory
. . .
myArray = null; //occupies almost no memory now
myArray[45] = 2003; //generates a "null pointer exception"
```

A major lesson here is that you can set any object equal to *null*.

Copying from array to array:

System.arraycopy*(theFromArray, fromIndex, theToArray, toIndex, howMany)* to **copy part of an array to part of another array**. The five parameters are explained as follows:

a. *theFromArray*…the array from which we are copying, i.e., the source.
b. *fromIndex*…the index in *theFromArray* from which copying starts.
c. *theToArray*…the array to which we will copy, i.e., the destination.
d. *toIndex*… the index in *theToArray* at which copying starts.
e. *howMany*…the number of array elements to copy.

If you have trouble remembering the order of from and to, just remember this little ditty, "From me to you."

Example:

```
char ch[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'};
char nn[] = {'1', '2', '3', '4', '5', '6', '7', '8'};
System.arraycopy(ch, 1, nn, 2, 3);
```

The destination array, *nn* will now look like this:
{'1', '2', 'b', 'c', 'd', '6', '7', '8'} *ch* array is unchanged.

Converting a *String* into a character array (and vice versa):

A *String* method we have not previously discussed is the ***toCharArray*** (signature: *public char[ ] toCharArray( )*) method. Here is how it's used:

```
char ch[];                //declared, but not initialized
String s = "ABCDE";
ch = s.toCharArray( );    //this initializes the ch array
```

Here's what the character array *ch* looks like now: **{'A', 'B', 'C', 'D', 'E'}**

It is also possible to reverse the process and convert character array *ch* directly into a *String* with:

```
String s = String.copyValueOf(ch);
//String.valueOf(ch) does the same.
```

There is another version of *copyValueOf* whose signature is:

```
static copyValueOf(char[]ch, int offset, int count)
```

Logical versus physical size of an array:

The **logical size** of the array in the following example is 5 since we only store numbers in the first 5 elements of this array. Notice the variable *max* in this particular example determines the logical size. The **physical size** (30 in this example) is always easy to determine. It's always *jk.length*;

```
int jk[] = new int[30]; //physical size… 30
int max = 5;

for (int j = 0; j < max; j++)
{
     jk[j] = j * 36;
}
```

The *Arrays* class:

This special class has some very useful methods that assist in the manipulation of arrays…especially **sorting**. For each of these methods we offer a description, the signature, and an example. To get these methods to work, you must **import** the *Arrays* class by putting ***import java.util.*;*** at the very top of your program. See Appendix I for more on the process of importing.

Sort:

Sort the array in ascending order (uses a merge sort…see Lesson 40).

```
public static void sort(int a[]) //Signature
```

Example:

```
int b[] = {14, 2, 109, . . . 23, 5, 199};
Arrays.sort(b);
//The b array is now in ascending order.
```

See the project at the end of this lesson where you will actually sort an array.

Binary search:

Perform a binary search (see Lesson 51) of an array for a particular value (this assumes the array has already been sorted in ascending order). This method returns the index of the last array element containing the value of *key*. If *key* is not found, a negative number is returned… *–k –1* where *k* is the index before which the *key* would be inserted.

```
public int binarySearch(int a[], int key)
//Signature
```

Example:

```
//Assume array b[] already exists and
//has been sorted in ascending order.
//The b array now reads
//{2, 17, 36, 203, 289, 567, 1000}.
int indx = Arrays.binarySearch(b, 203);
//search for 203 in the array
```

```
System.out.println(indx); //3
```

Equality:
Test for the equality of two arrays.

```
// Compares corresponding elements: true
// if the same…otherwise false.
public boolean equals(int a[], b[]) //Signature…
```

Example:

```
int x[] = {1, 2, 3, 4, 5};
int y[] = {1, 2, 3, 4, 5};
int z[] = {1, 2, 9, 4, 5};
System.out.println(Arrays.equals(x, y)); //true
System.out.println(Arrays.equals(x, z)); //false
```

Fill:
Fill an array with some specified value.
public void fill(int [], v) //**Signature**…fill array a with value v.
Example:

```
int pk[] = {1, 2, 3, 4, 5};

Arrays.fill(pk, 77);
//Array now looks like this {77, 77, 77, 77, 77}
```

String equivalent:
An entire array can be converted to a *String* similar to "[2, -3, 5, 18, 22]".
Example: Arrays.toString(myArray); //Typically printed as a test

The above discussion is for the *int* type arrays; however, all methods work for arrays of any of the primitive types and *String*s. The *sort* method works for objects from any class implementing the *Comparable* interface... All methods are *static*.

Command Line arguments:
Let's take a final look at the signature for the *main* method:

```
public static void main(String args[])
```

Now that we know about arrays, we can see that "*String args[ ]*" is declaring *args* as a *String* array. But where and how is this *args[ ]* array to be used? (Incidentally, this *args[ ]* array could be called by **any** legal variable name.)

The *args[ ]* array allows us to pass **command line arguments** to the *main* method. Entering a command line (see Appendix X) at the DOS prompt is one way to run a Java program. To do this you would need to be in a DOS console via the sequence Start | Run | *cmd* (don't use the older *command*) | OK):

```
java MyClass -46 Fleetwood.bat
```

What exactly does all this mean? The leading word *java* means to run the Java executable file (*java.exe*), *MyClass* (shown below) is the class containing the *main* method you wish to run, -46 is a *String* representing the first parameter we are passing (stored in *args[0]*), and *Fleetwood.bat* is a *String* representing the second parameter we are passing (stored in *args[1]*).

```
public class MyClass
{
     public static void main( String args[] )
     {
          System.out.println( args[0] ); //-46
          System.out.println( args[1] );
               //Fleetwood.bat
     }
}
```

Using a command line argument from the DOS prompt is a little awkward. Generally, you will need to first issue the command *cd C:\Program Files\Java\jdk1.5.0_04\bin* to change to the folder in which *java.exe* resides. (Your Java folder's name may be different.) You will also need to have compiled your class file (resulting in a file with extension *.class*) and have it stored in this same *bin* folder.

For users of the BlueJ Environment there is a much easier way to pass command line arguments. When you are ready to launch your *main* method, click on *void main(args)* and then in the resulting dialog, enter your arguments between the two braces as follows: {"-46", "Fleetwood.bat"}

Be sure to include the quotes. You can have as many arguments as you like. Many times, only two are used. It is customary to interpret those *String*s starting with a "-" as options and others as file names; however, as a programmer you may assign any desired meaning.

Using an array variable as an index:

Consider the following code that uses an array variable as an index for an array variable:

```
int ary[] = {5, 6, 7, 8, 9, 10};
System.out.println(ary[ ary[0] ]);
//10…ary[0] = 5, ary[5] = 10
```

The enhanced *for* loop ("for-each" style):

With the advent of Java 5.0 comes the much awaited "for-each" style of *for* loop. It is officially referred to as an **enhanced** *for* loop. Fundamentally, it lets us automatically loop through all the elements of a collection of objects, such as an array, from start to finish. This is done without specifying the length of the array and without an artificial, dummy integer index.

Traditional *for*-loop example:
This is illustrated below; first, by showing the traditional way of summing the squares of a sequence of numbers stored in array *x*:

```
int x[] = {4,3,2,1};
int sum = 0;
for(int j = 0; j < x.length; j++)
        sum = sum + x[j] * x[j];
System.out.println(sum);       //30… this is the problem 4^2 + 3^2 + 2^2 + 1^2
```

Enhanced *for*-loop example:

With the "enhanced *for*" style, the equivalent code would be:

```
//Equivalent code using the enhanced for method
int x[] = {4,3,2,1};
int sum = 0;
for(int varName: x)
     sum = sum + varName * varName;
System.out.println(sum); //30
```

Notice here in the parenthesis of the *for*-loop, *x* is the name of the object collection through which we wish to iterate, while *varName* is the local name given to it for use on each iteration of the loop. Thus, we can state the following syntax rule for the "enhanced *for*" style:

```
for(Type DummyName: ObjectCollectionName)
```

Read-only:

Unfortunately, the loop variable of the enhanced *for* loop is "**read-only**" with regard to *DummyName* in the example above, thus making its usefulness somewhat limited. This is illustrated by the following code in which we loop through all the elements of the *str ar*ray in which we "try" to change their values:

```
String str[] = {"one", "two", "three"};
for(String ss: str)
{ ss = "zero"; }
```

Beware: The expectation would normally be for all three elements of the *str* array to now equal "zero"; however, they remain the same. This is because the loop is read-only **with regard to *ss***. This code will compile and run; however, it accomplishes nothing. It should be noted, however, that direct references to the *str* array within the loop **would be** capable of changing the array.