Lesson 19…..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 <u>same</u> 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 <u>any object</u> 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 <u>from</u> which we are copying, i.e., the source.
b. *fromIndex*…the index in *theFromArray* from which copying starts.
c. *theToArray*…the array <u>to</u> 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 <u>from</u> and <u>to</u>, just remember this little ditty, "<u>From</u> me <u>to</u> 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 41).

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.

Exercise for Lesson 19

1. Write code that will create an array of 300 *BankAccount* objects. You are only to instantiate two of them. The object with index 47 should have a beginning balance of $92, and index 102 should have $1007. The name of your array will be *ba*.

2. Write an *if* statement that will decide if *k[3]* is equal to *jm[5]* where it is assumed that *k* and *jm* are numeric arrays.

3. Write an *if* statement that will decide if *s[2]* is equal to *ss[19]* where it is assumed that *s* and *ss* are *String* arrays.

4. Write an *if* statement that will decide if *cir[2]* is equal to *cirr[10]* (with regard to content) where it is assumed that *cir* and *cirr* are object arrays of type *Circle*.

5. What's wrong with the following code?
```
char months[];
months[0] = 'j';
```

6.
```
String suv[] = new String[20];
j = 0;
while(j < 17 )
{
        suv[j] = "Hello";
        j++;
}
```
What is the logical size of the *suv* array?
What is the physical size of the *suv* array?

7. Write code using *toCharArray* to convert *String d = "The quick brown fox jumped over the lazy dog."* into the character array *qbf*.

8.
```
double rub[] = {23.0,  -102.1,  88.23,  111,  12.02,  189.119,  299.88};
double dub[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```
Write a single line of code (using *arraycopy*) that will result in *dub* looking like this:
{1, 2, 3, 4, 111, 12.02, 189.119, 8, 9}

9. double[] zz, top = {12.1, 13.1, 14.1, 15.1, 18};
   zz = top;
   zz[2] = 99;
   top[3] = 100.2;
   Show what "both" arrays would look like at the completion of the above code.

10. char[] a, b;
    a = "Groovy dude".toCharArray( );
    b = "I like this".toCharArray( );
    System.arraycopy(a, 1, b, 0, 4);
    What do the two arrays look like at the completion of this code?

11. What must be true of any array before we can use *Arrays.binarySearch( )?*

12. Write code that will establish an array called *myArray* having the following elements, {189.01, 2000, -32, 56, 182, 2}. Then sort the array.

13. Assume the array *myArray* in #12 has been correctly sorted. What would be printed with the following?

```
System.out.println( Arrays.binarySearch(myArray, 56) );
System.out.println( Arrays.binarySearch(myArray, 102) );
```

14. What does the following print?

```
int xc[] = {123, 97, -102, 17};
int pk[] = {123, 79, -102, 17};
int gs[] = {123, 97, -102, 17};
System.out.println( Arrays.equals(xc, pk) + "\n" + Arrays.equals(xc, gs));
```

15. What does the following print?

```
int pickle[] = {1, 2, 3, 4, 5, 6, 7, 8};
Arrays.fill(pickle, -1);
System.out.println( pickle[4] );
```

16. If a command line reads, *java BigClass Munster Herman dude*, what will the following line inside the *main* method print?

```
System.out.println("Name=" + args[2] +args[1] );
```

17. What's printed by the following?

```
int px[] = {3, 4, 5, 6, 7, 8, 9};
System.out.println( px[   px[1] + 1   ]);
```

18. Write code using the "for-each" style of a *for* loop that will accumulate and print the product of the state variables *int jj* within each object of object array *objArray*. Assume the objects are created from the class *DummyClass.*

Arrays… Contest Type Problems

1. What is the value of *gem[1]* in the code to the right?

   A. –102
   B. 14
   C. 5
   D. 100
   E. –100

```
int [] gem = {-102, 14, 5, 100, -100};
```

2. Which code will sort the *gem* array in the code to the right?

   A. mergeSort(gem);
   B. Arrays.sort(gem[]);
   C. Arrays.sort(gem);
   D. Collections.sort(gem);
   E. Both C and D

3. What is the value of *g* when accessing the code to the right?

```
int [] stk = {1, 5, 19, 2, 20, 180};
int g = nerdStuff(stk) + 1;
```

   A. 3
   B. 2
   C. 0
   D. 7
   E. None of these

```
public static int nerdStuff(int [] cb)
{
        int counter = 0;
        for(int k=0; k<cb.length; ++k)
            if( cb[k] < 3 )
                ++counter;
        return counter;
}
```

4. Which of the following lines of code is a proper way to declare and initialize the *c* array?

   A. int [] c = new int[] {1, 2, 3, 4};
   B. int [10] c = {1, 2, 3, 4};
   C. int c = {1, 2, 3, 4};
   D. int[] c = new int {1, 2, 3, 4};
   E. Both A and B

5. What should replace **<*1>** in the code to the right in order that the *for*-loop variable, *j*, would cycle through all indices of the *a* array?

   A. j < a.length - 1
   B. j < a.length( )
   C. j <= a.length
   D. j < a.length + 1
   E. None of these

```
public static void testLoop(int [] a)
{
        for(int j=0; <*1>; ++j)
            ++a[j];

}
```

6. If **<*1>** has been filled in correctly in the code to the right, and *a[3] = 19* before calling *testLoop*, what is *a[3]* afterwards?

   A. 3
   B. 19
   C. 18
   D. 20
   E. None of these

7. What is output in the code to the right?

A. ancp
B. mbod
C. aocq
D. abcd
E. None of these

```
public class ArrayTest
{
   public static void main(String [] args)
   {
    String s1 = "abcdefghijk";
    char [] x = s1.toCharArray( );
    String s2 = "mnopqrstuvw";
    char [] y = s2.toCharArray( );
    int vv[] = {0,1,0,1};
    for(int j=0; j<vv.length; j++)  {
      switch (vv[j])
      {
            case 0:
               System.out.print(x[j]);
               break;
            case 1:
               System.out.print(y[j+1]);
       }
    }
   }
}
```

8. What is output in the code to the right?

A. 14
B. 15
C. 16
D. Throws an exception
E. None of these

```
public class ArrayTest
{
   public static void main(String [] args)
   {
    int a[] = {0,1,2,3};
    int b[] = a;
    int sum = 0;
    for(int j=0; j<3; j++)  {
         sum+=(a[j+1] * b[j]) + (a[j] * b[j+1]);
    }
    System.out.println(sum);
   }
}
```

9. What is output in the code to the right?

A. 102
B. 44
C. 56
D. Throws an exception
E. None of these

```
public class ArrayTest
{
   public static void main(String [] args)
   {
    int [] z1 = {2,3,4,5,6};
    int [] z2 = {1,2,1,2,1};
    double d = 0;
    for(int j=0; j<3; j++)
    {
     d = d + Math.pow(z1[j+1], 2)  +  Math.pow(z2[j], 2);
    }
    System.out.println(d);
   }
}
```

10. What is output in the code to the right?

A. 1002003007080
B. 1002007080500
C. 405030040080
D. 405060300400
E. None of these

```
public class MyTester
{
   public static void main(String args[])
   {
      int j, src =2, des=3, hm=2;
      int [] sa = {100,200,300,400,500};
      int [] da = {40,50,60,70,80};
      System.arraycopy(sa,src,da,des,hm);
      for(j=0; j<da.length; j++)
      System.out.print(da[j]);
   }
}
```

11. What is output in the code to the right?

A. 6
B. 1
C. 0
D. 2
E. Throws an exception

```
public class MyTester
{
   public static void main(String args[])
   {
      int [] aleve = new int[] {0,1,2,3,4,5,6,7,8};
      int n = 6;
      n = aleve[aleve[n]/2];
      System.out.print(aleve[n]%2);
   }
}
```

12. What replaces **<#1>** so that the product of all the elements in array *d* is returned?

A. for(double j: d) product *= d[j];
B. for(int j = 0; j < d.length; j++)
       product = product * j;
C. for(int j = 0; j < d.length; j++)
       product*= d[j];
D. for(double j: d) product *= j;
E. More than one of these

```
public static double getProduct( )
{
        double d[] = {100, -25, 16, 27, -102};
        double product = 1;
        <#1>
        return product;
}
```