Yoshi

CSC 205 Lab 3: More on Methods and Arrays

This short lab should be completed during class on Friday, September 16th.

Goals

After doing this lab, you should be able to:

- overload methods
- understand how arrays are passed to and returned from methods
- access two dimensional arrays in a number of specific ways
- begin using the Java `BigInteger` class

Materials Needed

Be sure you have the following on hand during the lab.

- This sheet and your course notebook.

Method

Change into your `Labs` folder (`cd Labs`), and make a `Lab3` folder (`mkdir Lab3`). Change into this directory to all work associated with this lab. Copy over the files you will need with this lab by typing :

```
cp /pub/digh/CSC205/Lab3/* .
```

Lab Steps

I. Passing One Dimensional Arrays as Parameters into Methods

1. The program `arOne.java` in your account demonstrates how we can send a one-dimensional array to a method and have its contents displayed. As this method simply prints out the array, its return type is `void`. Take a look at this program. A copy is included. We print the array of global constant size `ASIZE` out in `main` first. Notice in the statement :

```
ar1[ASIZE-1] = 2001;
```

does this set the last element of the array to 2001 or the next to last element? Set to last element

2. Now, let's do a complete trace through this program as it is written now to determine the output. Draw the contents of array `ar1` below.

0	1	2	3	4	5	6	7	8	9
11	12	13	14	15	16	17	18	19	20

3. Compile and test your program when you're done and check your results. Notice the differences between `Triple1` and `Triple2`. One is value-returning and the other is `void`. Notice how the `Triple2` method which is void has the right to actually modify the array object sent in as a parameter. The `Triple1` method makes a copy of the array, modifies the copy, and returns the copy.

II. Passing Two Dimensional Arrays as Parameters into Methods

1. The program `arTwo.java` in your account shows you how to declare a two dimensional array of integers and then use a method to print the contents of the array, `DisplayArray()`. Take a look at this program.

2. Notice the <u>second</u> method called `DisplayArray()`. The compiler will not mind at all that you have another method with the same name. Why not? What is it called if you have two methods with the same name in the same class scope?

Overloading

3. Now, what purpose exactly does this second version of `DisplayArray()` serve?

Allows the same method name to be reused when changing slightly different parameters

4. Now, let's do a complete trace through this program as it is written now to determine the output. Draw the contents of array `ar2` below.

10	11	12	13	14	15
16	17	18	19	20	21
22	23	24	25	26	27
28	29	30	31	32	33
34	35	36	37	38	39
40	41	42	43	44	45
46	47	48	49	50	-99

5. Compile and test your program when you're done and check your results.

6. Compile and run program `arThree.java`. The program accepts a series of numbers into an array and then calls a method to compute the sum. The program prompts you initially for the maximum number of items you want to enter. If you decided to enter fewer numbers, you just enter a negative value and the input is terminated.

III: Simple factorials

1. The program `factorial.java` will compute the factorial of an integer value. Take a look at it. Now, let's compile and run this program using a single integer as <u>a command line argument</u>. For example, `java factorial 13`.

2. Factorials grow very quickly. Compute the values below:

13!	1932053504
14!	1278945280

wrong

3. This program can only compute factorials up to 12!. That's not very useful. Let's try changing the type of the result to get larger values in the final part of lab.

IV: The BigInteger Class

cp factorial.java bifactorial.java

1. To go beyond the primitive types, we need to use a new class. Fortunately, the `BigInteger` class is one of the standard Java classes, so to get extremely large integers, we can use this class. Copy the program `factorial.java` to `bifactorial.java` using the `cp` command. Now, go in and change the class name to `bifactorial` and everywhere you have `int` should be changed from `int` to `BigInteger`.

2. Changing to a non-primitive value takes a bit of doing. First of all, you need to include the library at the top of the program with
   ```
   import java.math.BigInteger;
   ```

3. Second, in the `fact` class method, there is no direct conversion between integers and BigIntegers, so you'll need to replace `product = 1;` with
   ```
   BigInteger product = BigInteger.ONE;
   ```

   If you are using an integer value other than 0 or 1, you can create a new BigInteger with its value using `BigInteger.valueOf (the integer)`. So, you could have initialized the product using
   ```
   product = BigInteger.valueOf(1);
   ```

4. Finally, you can not use the standard mathematical symbols +, -, *, etc on new types. (Some programming languages do allow you to redefine these, but Java is not one of those languages.) Instead of `product *= i;` you'll need
   ```
   product = product.multiply (BigInteger.valueOf(i));
   ```

5. Get your program to compile and try a run that computes the factorial of a large number (somewhere between 200 and 1000).