CSC 204 Lab 6: Numbers, Primitive and Fancy

Chapter 4 of the text covers Java's primitive types. In this lab, you'll see how the differences in the ranges of these types affect a program. You'll have a chance to try out some of the methods in the `Math` libraries. You'll also see how to use the `BigInteger` library.

Goals

After doing this lab, you should be able to:

- Import projects into Eclipse and edit, compile and run them.
- Copy files in Eclipse.
- Discuss the boundary limits of the types `short`, `int`, `long`, `float`, and `double`.
- Find the limits on the primitive types from the appropriate methods in `java.lang`.
- Describe the rounding options of the `Math` class.
- Use the basic features of `BigInteger`.

Lab Preparation

Read through this lab. Review the material in Chapter 4. You may find the API for the `Math` and `BigInteger` classes useful.

Materials Needed

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

- This sheet,
- Your textbook, and
- A computer with Internet access to our shared folder on Blackhawk.

Lab Setup - Starting Eclipse

- Make a copy of this Lab6 document. You should edit your copy with answers asked within, and then place it in our shared Google folder for this lab when complete.
- Open Eclipse and select your Eclipse workspace on Orion.
- Create a new Java Project named “Lab6”.
- Copy the three Java files from our Blackhawk-Lab6 folder into the src folder of your Eclipse Lab6 project: Factorial.java, NumberLimits.java, and RoundingOff.java.

Part 1: Simple factorials

The program Factorial.java will compute the factorial of an integer value. Take a look at it.

Use the program to find 10!

Factorials grow very quickly. Compute the values below:

10!	3628800
11!	39916800
12!	479001600
13!	1932053504

Something very peculiar has happened…13! is incorrect . This is because 13! is too large to be represented as an integer, so the leading binary digit has been removed.

What exception was raised when the number became too large to be stored?

An integer overflow occurs

What's peculiar about 20!?

20! = -2102132736. It's a negative number.

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.

Part 2: Bigger factorials

Ah! Maybe changing the result type from `int` to `long` will fix the program. Create a new class called LongFactorial.

Copy the program `Factorial.java` to `LongFactorial.java` (just copy and paste from one editor tab into the other). Now, go in and change the name to LongFactorial and everywhere the result is declared from `int` to `long`. (The three places you need to do this are labeled in the code. Be sure you understand why these are the three places that need to change.)

Compile and run the program. Check the values for 15! and 16! . Keep trying larger values until you see a problem (don't worry, it won't take long!).

Give the values and results here:

```
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = -4249290049419214848
```

Has changing the type to long helped much?
Yes, we are now able to see a much higher correct value.

Part 3: Even bigger factorials

So, widening to a long doesn't help much. Maybe changing the result type from `long` to `float` will let us compute larger factorials. We'll have to sacrifice some digits of precision, but the extra range may be worth it. Copy the program `LongFactorial.java` to `FloatFactorial.java`. Now, go in and change the name to `FloatFactorial` and everywhere the result is declared from `long` to `float`.
Compile and run the program.
Compare the values for 15! and 16! with those given by LongFactorial.

```
Long Factorial
15! = 1307674368000
16! = 20922789888000

Float Factorial
15.0! = 1.30767428E12
16.0! = 2.09227885E13
```

When do you get fewer digits of precision from FloatFactorial than from LongFactorial?

Keep trying larger values until you see a problem (don't worry, it won't take long!).

Give the values and results here: It starts giving fewer digits at 21!

Since the floating point numbers have a representation for infinity, we get that instead of overflow values when using floating point numbers.

Part 4: Still bigger factorials

We are rapidly running out of types, but let's try doubles. You got it, copy the program `FloatFactorial.java` to `DoubleFactorial.java`. Now, go in and change the name to `DoubleFactorial` and everywhere the result is declared from `float` to `double`.
Compile and run the program.
Now how large a value can you compute? (You'll want to step by more than 1 in this case. For example, you might want to try 50!, 100!, 150!, etc. to help find the largest value.)

```
170.0! = 7.257415615307994E306
```

Change the program to work with bytes and shorts.
Summarize all of your results in the table below.

type	Largest n	n!
byte	9!	-128
short	17!	32768
int	33!	-2147483648
long	65!	-9223372036854775808
float	34.0!	2.9523282E38
double	170.0!	7.257415615307994E306

While the type `double` is an improvement over the others, there are still a lot of integers we cannot compute the factorial for. We also lose digits of precision when we use floating point types. To improve this, we'll have to move beyond the primitive types. Before doing that, let's look at the other primitive types a bit more closely.

Part 5: Limits to types

Since the types in Java have fixed lengths, they have predetermined maximum and minimum values. You can get these from the respective classes, so the largest value of a `byte` is `Byte.MAX_VALUE`.

Edit the program `NumberLimits.java` to print the minimum and maximum values of each of the numeric types. Fill in the table below.

type	Smallest value	Largest Value
byte	-128	127
short	-32768	32767
int	-2147483648	2147483647
long	-9223372036854775808	9223372036854775807
float	1.4E-45	3.4028235E38
double	4.9E-324	1.7976931348623157E308

Part 6: Rounding off with the math library

There are many different ways to convert floating point numbers to integers. Using the program in `RoundingOff.java`, fill in the table below with the different values that are returned.

Expression	Value
`round(3.4)`	3
`round(3.5)`	4
`round(3.6)`	4
`ceil(3.4)`	4.0
`ceil(-3.4)`	-3.0
`floor(3.6)`	3.0
`floor(-3.6)`	-4.0
`rint(3.5)`	4.0
`rint(4.5)`	4.0
`rint(3.2)`	3.0
`rint(-3.2)`	-3.0
`rint(-3.8)`	-4.0

Part 7: Biggest factorials

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 `BigFactorial.java`. Now, go in and change the name to `BigFactorial` and everywhere the result is declared from `int` to `BigInteger`.
Try compiling this program. You should notice a lot of errors. 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;
```

Second, there is no direct conversion between integers and BigIntegers, so you'll need to replace `product = 1;` with

```
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);
```

Finally, you cannot 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));
```

Get your program to compile and run correctly. You can test it on smaller values before trying it on large values.

Deliverables

When you are finished with this lab, upload the src folder from this Eclipse project to your shared Google folder. Change the name of the uploaded folder to “Lab6”. Place your copy of this document, with your answers, into this shared Lab6 folder. Enjoy, be creative, and have fun!