Math examples

Purpose: Learn how to do basic mathematics in java. Learn about `if` statements and *loops*. Learn some more about classes and functions.

I. **Create the Math Examples project.** Create a new NetBeans project named "MathExamples" with main class named "Factorial". Here are the step by step instructions:
   A. Click on the *File* menu, then select *New Project*.
   B. A new window will pop up. On the first page select "Java" from *categories*, and "Java Application" from *projects*. Then click *next*.
   C. Set *Project Name* to "MathExamples." Select a *Project Location* and *Project Folder*.
   D. Make sure *Create Main Class* is checked. Enter "Factorial" into the adjacent textbox. Click *finish*.
   E. The project should be created, and the file "Factorial.java" should open.

II. **Factorial.java** Our program "Factorial.java" will read an integer *n* from input, then output *n!*.
   A. We will place all the code for "Factorial.java" in the `main` method:

```
public static void main(String[] args) {...}
```

   We will describe the content of the main function in steps below, and then summarize at the end. Everything in statements B-E will go in the `main` method.

   B. We read an integer from the command line with the line:

```
int n=Integer.parseInt(args[0]);
```

   Comments:
   1. The variable `args` stores an array (or list) of Strings. Writing `args[0]` selects the first string in the array. Hopefully, it is something like "5".
   2. An `int` is a primitive data type which stores an integer whose absolute value is less than 2 trillion or so.
   3. The class `java.lang.Integer` is a helper class for dealing with ints. In particular it has a function `parseInt`, which converts a `String` to an `int`.
   4. The integer is stored in a variable named `n`.

   C. Since `n!` is undefined when `n` is negative, we will check to see if `n` is negative. If it is negative we will print an error then exit. We do this with the following code:

```
if (n < 0) {
    System.out.println("The number (" + n + ")! is undefined because " + n + " is negative.");
    System.exit(0);
}
```

   Comments:
   1. The two lines in the brackets will be exectuted exactly when n is negative.
   2. Adding a number has the effect of converting `n` to a string and then concatenating the strings.
   3. The line `System.exit(0);` terminates the program.

   D. Now we will compute n!. To do this, we begin by defining two new integers:

```
int factorial=1;
int i=1;
```

   The number `factorial` will eventually store `n!`. We will iterate `i` from `1` to `n`, multiplying `factorial` by `i` each time:

```
while (i <= n) {
    factorial = factorial * i;
    i = i+1;
}
```

   Comments:
   1. When the program reaches a while statement, it immediately checks to see if the

expressions enclosed in parentheses is true. If it is true, then it executes the statements enclosed in the brackets `{...}`. Otherwise, the program continues to the next statement after the brackets.

2. Whenever the program reaches the end of the statements in the brackets, it again checks the expression in parenthesis. If the expression is true, it executes the statements again. Otherwise, it moves on.
3. Our loop exits when `i=n+1`.

E. Finally, we will print the result of our computation so that the user can see it. We use the following statement:

```
System.out.println( "" + n + "! = " + factorial);
```

The construction `""+n` takes the empty string `""` and concatenates the string formed my converting the integer `n` to a string. Then we concatenate `"! = "` and `factorial` converted to a string. We print something like `"5! = 120"`.

The entire `main` method is included below:

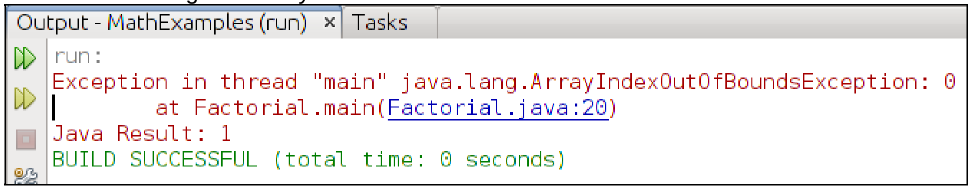
```
public static void main(String[] args) {
     int n=Integer.parseInt(args[0]);

     if ( n < 0) {
         System.out.println("The number (" + n + ")! is undefined because " + n + " is negative.");
         System.exit(0);
     }

     int factorial=1;
     int i=1;

     while (i <=n ) {
         factorial = factorial * i;
         i = i+1;
     }

     System.out.println( "" + n + "! = " + factorial);
}
```

You can also view the whole file with comments here: Factorial.java.

III. Running Factorial.java

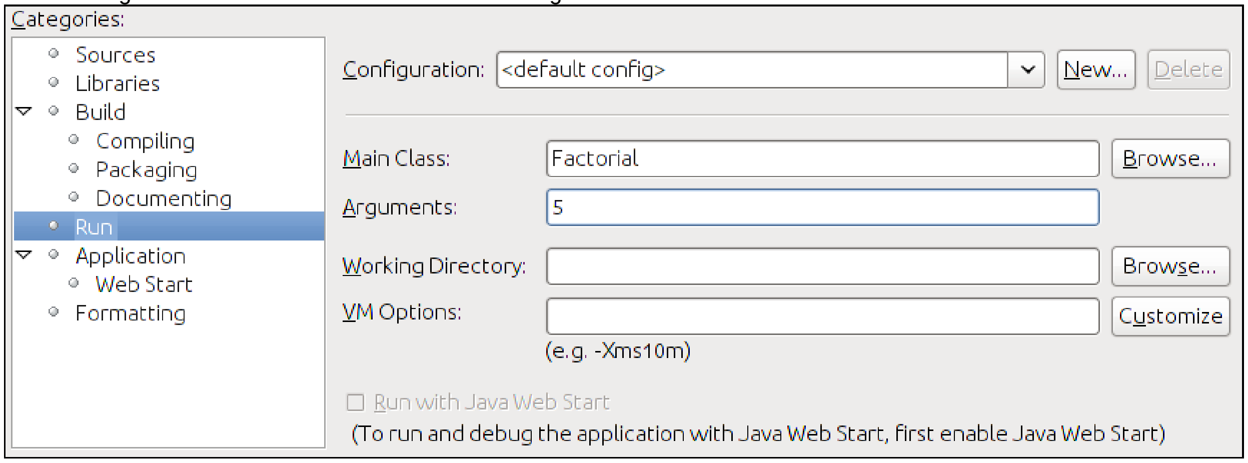
A. Running via NetBeans:

1. Using the menus select "Run > Set Main Project > MathExamples."
2. If you naively click menus "Run > Run Main Project", you get an error because there are no command line arguments by default.



```
run:
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at Factorial.main(Factorial.java:20)
Java Result: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

3. To solve this select menus "File > Project Properties". A window will pop up. Select "Run" from "Categories." Enter a number like "5" under arguments. The window should look like:



Configuration: <default config>
Main Class: Factorial
Arguments: 5
Working Directory:
VM Options:
(e.g. -Xms10m)
(To run and debug the application with Java Web Start, first enable Java Web Start)

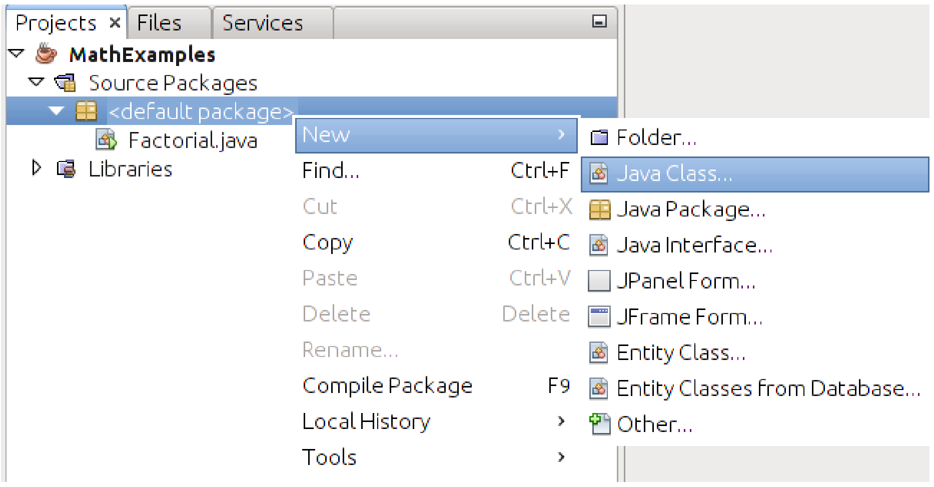
4. Now "Run > Run Main Project" should work.

B. Running on the command line:

1. From the NetBeans menu select "Run > Clean and Build Main Project."
2. Now if you open a terminal and go to the "dist" folder of your NetBeans project, there should be a file named "MathExamples.jar".
3. Run the java program using the command:
   ```
   java -jar MathExamples.jar 10
   ```
   The "10" here is the command line argument. The output should be:
   ```
   10! = 3628800
   ```

IV. **Exp.java.** We will make a new program "Exp.java." The program will read a real number *x* from the command line. Then it will compute an approximation of *exp(x)*.

A. **Steps to create the file "Exp.java":**



1. Switch to the Projects tab in the top left subwindow of netbeans.
2. Make sure "MathExamples > Source Packages > <default package>" is open. See the screen shot at right.
3. Right click on "<default package>", then select "New > Java Class".
4. A new window will pop up. Set "Class Name" to "Exp". Then click "Finish".
5. A source code subwindow should open labeled "Exp.java".

B. **The factorial method.** Inside the "Exp" class we will create a factorial method of the form:

```
public static int factorial(int n) {...}
```

The function takes as input the integer `n`. (Input is in parentheses). The `int` after `static` indicates that the function will return an int. The complete function is given by:

```
public static int factorial(int n) {
    int factorial=1;
    for (int i=1; i<=n; i++) {
        factorial=factorial * i;
    }
    return factorial;
}
```

The only new aspect of this code is that we used a `for` loop rather than a while loop. In the parentheses after the for loop, there are three expressions: The first, `int i=1`, is run when the loop begins, and creates an integer i with value 1. The second expression `i<=n` returns true or false. This has the same purpose as the expression in the while loop; the statements in brackets `{ ... }` will be evaluated so long as this expression is true. The third expression `i++` is evaluated after the statements in brackets, whenever they are evaluated. The expression `i++` is equivalent to `i=i+1`. So a `for` loop is just a compact form of the while loop.

C. **The power method.** We will also create another function:

```
public static double power(double x, int n) {...}
```

This function takes two inputs, a `double` named `x` and an `int` named `n`, which we assume is non-negative. A double stores an approximation of a real number. The function will also return a real number. The number returned will be x^n. The code for this follows:

```
public static double power(double x, int n) {
    double ret=1.0;
    for (int i=1; i<=n; i++) {
        ret = ret * x;
    }
```

```
    return ret;
}
```

You should be able to figure out how the function returns x^n.

D. **The main method.** Recall that our goal was to evaluate *exp(x)*. We will do this in our `main` method.

   Our method will do the following. It will read a real number `x` from input, then it will output `exp(x)`. We use the Taylor series for `exp(x)`:

   $$exp(x)=1+x+x^2/2+x^3/6+...+x^n/n!+....$$

   The source code for our main function is given below:

```
public static void main(String[] args) {
    double x=Double.parseDouble(args[0]);
    double sum=0;
    for (int i=0; i<15; i++) {
        sum=sum+power(x,i)/factorial(i);
    }
    System.out.println("exp("+x+") = "+sum);
    System.out.println("A second opinion: exp("+x+") = "+Math.exp(x));
}
```

   Comments:

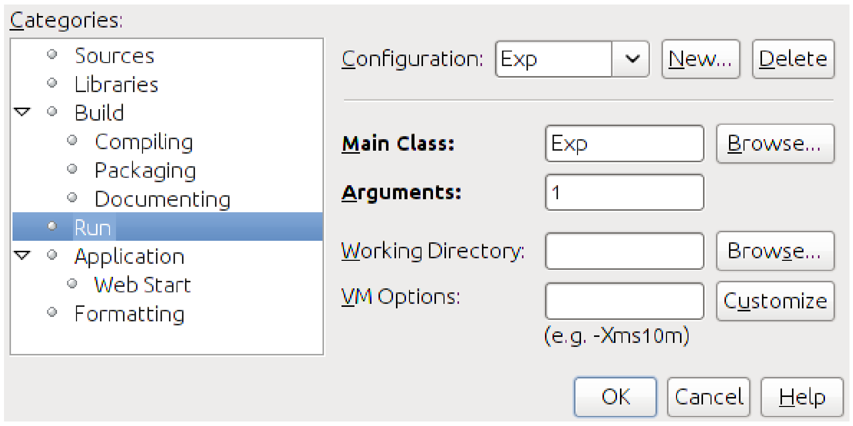
   1. `Double` is a helper class for doubles. The `parseDouble` function converts a string to a double.
   2. We call our `factorial` and `power` functions repeatedly in the loop.
   3. Java has a class `java.lang.Math` class which implements many mathematical functions including `exp`.

You can view the commented source code file: Exp.java.

V. Running Exp.java

A. **Running via NetBeans.** Steps to run the program in NetBeans:

   1. Select menus "File > Project



Configuration: Exp
Main Class: Exp
Arguments: 1
Working Directory:
VM Options:
(e.g. -Xms10m)

      Properties". A window will pop up. Select "Run" from "Categories."
   2. To the right of "Configuration" click "New". Enter "Exp" as the "Configuration Name" and click okay.
   3. Now with "Exp" selected next to configuration, enter "Exp" for the Main class (you can use the browse button if you want) and enter "1" under arguments.
   4. This window should look as on the right. Click okay to close the window.
   5. Run the program with the menus "Run > Run Main Project," or simply press F6.

B. **Running on the command line.**

   1. From the NetBeans menu select "Run > Clean and Build Main Project."
   2. Now if you open a terminal and go to the "dist" folder of your NetBeans project, there should be a file named "MathExamples.jar".
   3. Run the java program using the command:
      ```
      java -jar MathExamples.jar 1
      ```
   4. Because the program "Factorial.java" is also included in the jar, you can also run this program. Do so with the command:
      ```
      java -cp MathExamples.jar Factorial 7
      ```

VI. Limits on primitive types.

A. The types "int" and "double" store numbers which take up a fixed amount of size on the computer, and so can not store numbers of arbitrary size. For instance, if we run our Factorial program with argument "20", then our program reports "20! = -2102132736." A negative number!

B. Fortunately, Java has a built in class java.math.BigInteger to work with integers of arbitrary size.

C. We will demonstrate the BigInteger class by writing a factorial program using it.

D. Java also has a built in class, java.math.BigDecimal, which does arbitrary precision arithmetic.

VII. BigFactorial Example

A. Create a java class in the MathExamples project named "BigFactorial". (Follow the directions used for the class Exp, if you forgot how.)

B. Open "BigFactorial.java" for editing.

C. Because we want to use the BigInteger class, we need to add the following line:

```
import java.math.BigInteger;
```

```
import java.math.BigInteger;
```

This line must appear somewhere above the class definition (`public class BigFactorial {...}`).

D. Inside the BigFactorial class add the following method:

```
public static BigInteger factorial(int n) {
    BigInteger factorial=BigInteger.valueOf(1);
    for (int i=1; i<=n; i++) {
        factorial=factorial.multiply(BigInteger.valueOf(i));
    }
    return factorial;
}
```

E. The API (Application Programming Interface) specification is a very useful resource. You can view the documentation for BigInteger here. We are using only two functions from the BigInteger class and they are well documented:

1. The valueOf method
2. The multiply method

Learn to read and understand the API. Once you can do this, you can teach yourself java!

F. From the API, we see the method declaration for `valueOf` is:

```
public static BigInteger valueOf(long val)
```

Comments on the method declaration:

1. The declaration has two modifiers:
   i. The word `public` just means that we can call the function.
   ii. The word `static` means that to call the function we refer directly to BigInteger, e.g. `BigInteger.valueOf(7)`.
2. The word `BigInteger` indicates the return type; the function will return a BigInteger.
3. The word `valueOf` indicates the name of the function.
4. The stuff in the parentheses is the parameter list. This method takes one parameter of type `long`. The `long` type is a primitive data type which stores an integer, which can be slightly bigger than an `int`. An `int` can be converted directly to a `long`, and so this function can be called with an `int` as well.

G. From the API, we see the method declaration for `multiply` is:

```
public BigInteger multiply(BigInteger val)
```

The absence of the modifier `static` means that to call the function you need to use an object to call it. In our code, we use:

```
factorial.multiply(BigInteger.valueOf(i))
```

Here, `factorial` is an object of class BigInteger. We use it to call multiply with parameter `BigInteger.valueOf(i)`, which is a newly constructed BigInteger with the same value as `i`. The function will return the product of the two BigInteger objects involved.

H. Our main method will be:

```
public static void main(String[] args) {
    int n=Integer.parseInt(args[0]);
    System.out.println(""+n+"! = "+factorial(n));
}
```

This code simply converts the first argument to an integer, then prints the result returned from the `factorial` method.

I. You can run this program by following the instructions used to run "Exp.java".
J. Complete source code for BigFactorial.java without comments:

```
import java.math.BigInteger;

public class BigFactorial {

    public static BigInteger factorial(int n) {
        BigInteger factorial=BigInteger.valueOf(1);

        for (int i=1; i<=n; i++) {
            factorial=factorial.multiply(BigInteger.valueOf(i));
        }

        return factorial;
    }

    public static void main(String[] args) {
        int n=Integer.parseInt(args[0]);
        System.out.println(""+n+"! = "+factorial(n));
    }
}
```

You can see the commented source code here: BigFactorial.java.

Useful Links:

- Parts of the Oracle's Java Tutorial:
  - Discussion of primitive types.
  - Discussion of methods.
  - Arrays: in case you want to do more with the argument array passed to the main method.
- Portions of the Java API:
  - java.lang.Math API: This class comes with many commonly used mathematical functions such as trigonometric functions, for instance.
  - java.math.BigInteger API: Handles arbitrarily large integers.
  - java.math.BigDecimal API. Handles arbitrary precision floating point arithmetic.