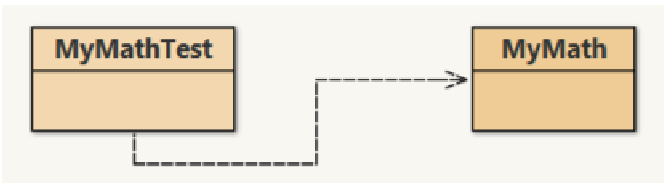
Challenge:	*#4*
Title:	***MyMath (Methods, Selection and Iteration)***
Marks:	*5% of module marks*
Partner:	*Complete with a partner, and only one submission.*
Objectives:	• *Design and code a class (**MyMath**) and a test class (**MyMathTest**).* • *Explore parameter passing.* • *Explore the **if**, **while** and **for** control statements.* • *Use **javadoc** to document each method.*
Submission:	*Submit to Sulis Assignments by Wed Week 11, 10 Dec, 23:00* ***Directly** submit the java files **MyMath.java** and **MyMathTest.java**.* ***Do not submit the java files within a zip file, as it will not be corrected.*** *Ensure that an appropriate file header comment is included in each java source file with author name(s), id number(s), last date of modification and a short description of the java class.*
Notes:	*All module handouts and laboratory/challenge work should be maintained in an accessible file storage device.* *The work completed should be available in a folder named **Challenge4**.* ***Reminder: Maintain regular backups of all your work.***

1. Exercise: Two classes

The exercise is to create two classes, a class **MyMath** and a test class **MyMathTest**:



2. MyMath

a) Create a class called **MyMath** that will contain a set of **methods** to calculate various mathematic numeric operations.

Each method should return a single value, and should not print anything.

There should be no **instance variables** in the **MyMath** class.

Some methods will correspond directly to mathematical functions, see definition of a function at:

http://en.wikipedia.org/wiki/Function_(mathematics)

There is more information on Java methods in notes and at

https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html

b) Each **method** must be documented with appropriate **javadoc** comments. Check that the comments are ok by invoking **javadoc**

(**BlueJ: Tools > Toggle Documentation View**; ***NetBeans: Run** > **Generate Javadoc***)

Use the method template below which outlines the structure of a javadoc comment:

```
/**
  * Short description of the method
  *
  * @param par1  Short description of the first parameter      Required only if
  * @param par2  Short description of the second parameter     there are parameters
  *   ...
  * @return  Short description of the returned value          Required if not void
  */
public static returnType methodName( type1 par1, type2 par2, ... )
{
   returnType result;
   // the code that calculates the required result
   return result;
}
```

c) Test the class **MyMath** methods by writing a test application in a class named **MyMathTest** that has a **main** method, which tests each method by invoking it at least once with an appropriate test case.

For each method, implement at least one test case in the **MyMathTest** class. You can implement all test cases in **main**. Otherwise, consider using a separate method for each test and call it from **main**.

Each test case should have a unique identifier name, purpose, inputs and expected result, and include a descriptive comment in the code just before the test code: by including a comment similar to below.

```
/*
 * Name:
 * Purpose:
 * Input Parameters:
 * Expected Result:
 */
```

For example, the test definition for the **average** method, which returns the average of **a**, **b**, **c** and **d**:

double average(double a, double b, double c, double d):

```
/*
 * Name:                averageTest
 * Purpose:             Test the class MyMath average method
 * Input Parameters:    a = 1.0, b = 2.0, c =4.0, d=8.0
 * Expected Result:     3.75
 */
```

For each test case, the code should print the test name, purpose, method name, the actual value of the arguments, the result and an indication if the result is as expected (i.e. if the test passed or failed).

```
averageTest: Test the class MyMath average method
MyMath.average(1.0, 2.0, 4.0, 8.0)    Result is 3.75
Test passed
```

d) To call a method, specify the class name, the method name and arguments as follows:
*(all methods are **public static** you can call them directly, no need to create a MyMath object)*:

```
MyMath.methodName( arg1, arg2, ...);
```

3. Code and test all of the following methods [5 marks]

All statements in the method bodies below must only use the parameters, local variables, operators, control statements and other **MyMath** methods.

Function:

a) **double average(double a, double b, double c, double d)** that returns the average of the four **double** parameters **a**, **b**, **c** and **d**.

$$\frac{1}{4}(a+b+c+d)$$

b) **long factorial(int n)** `to` calculate $n!$ Use this code or improve it:

```
/**
 * Calculate n factorial (n!), n less than 21.
 * @param n the parameter whose factorial is required
 * @return the factorial of the parameter
 */
public static long factorial(int n)
{
     long result = 1;
     int counter = 2;
     while (counter <= n)
          {
        result = result * counter;
        counter++;
     }
     return result;
}
```

$$n! = \prod_{i=1}^{n} i$$

c) **int max(int a, int b, int c)** that returns the maximum of the three **int** parameters **a**, **b** and **c**. Use an **if** statement.

max(a, b, c)

d) **double max(double a, double b, double c)** that returns the maximum of the three **double** parameters **a**, **b** and **c**.

max(a, b, c)

Note the method name **max** is the same as the method in part b), this is known as ***Method Overloading***.

e) **int sum(int n)** to calculate the sum of the first n natural numbers from 1 to n i.e. **1 + 2 + 3 + … + n**.

$$\sum_{i=1}^{n} i$$

f) **long biCoeff(int n, int k)** to calculate binomial coefficients also known as “n choose k”. Use the above factorial method e.g. **factorial(n)/factorial(k)/factorial(n-k)**.

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

g) **double power(double a, int b)** that returns the value of the first parameter **a** to the power of the second parameter **b**.

a^b

The power method must not call any other methods (use a loop, note that **b** is an integer and may be negative). Note the type of the second parameter is **int**.

4. Code and test additional methods [2 marks]

h) **Each author** must write at least 2 other methods to implement mathematics based formulas. Each method will normally accept one or more formula parameters, evaluate and return the result. Include author's name in the method comment.

One method can be basic and one should be challenging. See examples on the next page and feel free to rename (use meaningful names).

Implement the methods in the **MyMath** class and test in the **MyMathTest** class. Use a number of test cases if necessary. The same method template discussed earlier applies also to your own methods.

Examples of basic methods to implement and test

i. **double convertCelsiusToFahrenheit(double celsius)** that converts a value in degrees Celsius to Fahrenheit. The relationship is given by the equation:

$$F = \frac{9}{5}(C) + 32$$

Where F is the temperature in Fahrenheit, and C is the temperature in degrees Celsius provided as an input parameter to the method.
Note: be careful with using integer constants in Java as 9/5 is 1 (integer result only) not 1.8.

ii. **int countNumberOfRealRoots(int a, int b, int c)** that computes the discriminant D of a quadratic equation $ax^2 + bx + c = 0$ and returns the number of roots given by:

$$D = b^2 - 4ac$$

If D > 0, the quadratic equation will have 2 roots. If D = 0, the quadratic equation has one root. Otherwise, if D < 0, the equation has no real roots. In all cases, the method must return the number of real roots. There is no requirement to return the values of these roots.

Verify your implementation works as expected by using 3 equations as input arguments to the method: e.g. $3x^2 + 7x + 9 = 0$ [0, D<0] $4x^2 - 12x + 9 = 0$ [1, D=0] $7x^2 - 5x - 1 = 0$ [2, D>0]

iii. sum of squares/cubes

iv. basic equations/functions from your other modules, …

Examples of more challenging methods to implement and test

i. **double computeWindChillTemp(double ambientTemp, double velocity)** that computes the wind chill temperature, given values for ambient temperature and velocity
You can use the following equation:

$$T_{wc} = 13.12 + 0.6215T_a - 11.37v^{0.16} + 0.3965T_a v^{0.16}$$

Where T_{wc} and T_a are the wind chill temperature and ambient temperature in degrees C respectively, and v is the wind speed in km/hr. Use the Java class **Math pow** method for calculating $v^{0.16}$.

ii. **double computeDrugConcentrationAtTime(double hoursAfterDose)** which computes the concentration of a drug in the body C_p, any time (in hours) after the dosage has been administered:
The plasma concentration of the drug C_p can be modelled by the equation assuming fully absorbed:

$$C_p = \frac{D_G}{V_d}\frac{k_a}{(k_a - k_e)}(e^{-tk_e} - e^{-tk_a})$$

Where D_G is the dosage administered (mg), V_d is the volume distribution (L), k_a is the absorption rate constant (h^{-1}), k_e is the elimination rate constant (h^{-1}) and t is the time (h) since the drug was administered. For a certain drug, the following quantities are given: D_G = 150mg, V_d = 50L, k_a = $1.6h^{-1}$, k_e = $0.4h^{-1}$. Use the Java class **Math exp** method to calculate e^{-tk_e} and e^{-tk_a}

iii. **double calcPi(int numberOfTerms)**

which will estimate the value of π using **n** terms from the infinite series (Gregory–Leibniz series):

$$\pi = 4 - 4/3 + 4/5 - 4/7 + 4/9 - 4/11 + \ldots$$

where the parameter **numberOfTerms** specifies the number of terms to calculate and sum.

iv. calculate compound interest given principal amount, years and interest rate

v. determine if a number is a prime,

vi. more advanced equations/functions from your other modules, …