# Session 1 - Basics

## Exercise 1:

You'll now create a program that computes the distance an object will fall in Earth's gravity from an initial position.

### Part One

1. Under your session1.introduction package, create a new class called `GravityCalculator`.

2. Copy and paste the following initial version (you can align your code with *Ctrl-shift-F* or *command-shift-F* for Mac):

```
package session1.introduction;

class GravityCalculator {
     public static void main(String[] arguments) {
          double gravity = -9.81; // Earth's gravity in m/s^2
          double initialVelocity = 0.0;
          double fallingTime = 10.0;
          double initialPosition = 0.0;
          double finalPosition = 0.0;
          System.out.println("The object's position after " +
               fallingTime + " seconds is " +
               finalPosition + " m.");
     }
}
```

3. Run it in Eclipse (Run → Run As → Java Application).

What is the output of the unmodified program?

### Part Two

Modify the example program to compute the position of an object after falling for 10 seconds, outputting the position in meters. The formula in Math notation is:

$$x(t) = 0.5 \times at^2 + v_i t + x_i$$

| Variable | Meaning | Value |
|---|---|---|
| $a$ | Gravity, acceleration ($m/s^2$) | -9.81 |
| $t$ | Falling Time (s) | 10 |
| $v_i$ | Initial velocity (m/s) | 0 |
| $x_i$ | Initial position | 0 |

*Note*: The correct value is -490.5 m.

## Exercise 2:

A corporation needs a program to calculate and employee's pay per week. The Department of Labor requires that employees get paid one and half times their regular pay for any hours over 40 that they work in a single week. For example, if an employee works 45 hours, they get 5 hours of overtime, at 1.5 times their base pay salary. The department also requires that employees be paid at €9.61/hour (minimum wage). The corporation requires that an employee does not work more than 60 hours in a week.

**Rules:**
- An employee gets paid (hours worked) × (base pay), for each hour up to 40 hours.
- For every hour over 40 hours, they get overtime pay (1.5 x base pay).
- The base pay cannot be less than €9.61/hour (minimum wage). If this happens, print an error message.
- A person can work a maximum 60 hours a week. If this happens, print an error message.

**Instructions:**
In the same package, create a new class called `Pay`.
Your class should have a method called `printPay` that calculates and prints someone's weekly pay. The method should take as input the number of hours worked in the week, and the base pay.

Add in your main function a few example calls to pay, trying to best test if it behaves correctly (i.e., try input arguments that test all the paths of your conditional statements).

**Hint:** Do not try to write the entire program in one go. It is much easier to write a small piece and test it, then write another small piece and test it. For example, start by writing just a skeleton of your method and your main program. Then add the code to do the normal salary computation, without any special rules. Then add each additional rule, one at a time. You should test your program with simple test inputs to check that you handle each case.

## Exercise 3:

Create a new class called `NaturalNumbers` who will be responsible for printing natural numbers and their sum. In the class add a method `printNN` that takes as input an integer *n* and prints the first *n* natural numbers, their squares, their sum and their average. From your main method call your `printNN` method for different inputs.

**Hint:** Do not try to write the entire program in one go. It is much easier to write a small piece and test it, then write another small piece and test it. For example, start by writing just a skeleton of your method `printNN` that only returns the number *n* to make sure you can call it properly from your main method. Then try to return the number *n* and its square. Then add a *for* loop to calculate the averages (consider using a storage variable). You should test your program with simple test inputs.

## Exercise 4:

Write a class `Triangle` that prints a right angle triangle with numbers which will repeat in a row as many times as the number value. For example an input of *4* will create a triangle for the numbers 1-4:

```
1
22
333
4444
```

**Hint:** consider using the methods System.out.print and System.out.println.

## Exercise 5:

With ten of your friends you participate in the Paris Marathon. You'll create a program for outputting some running times. Create a new class called Marathon. In your main function, make up 10 running times that are between 150 min (2.5h) and 300 min (5h) and store them in an array.

1. Find the best time.
2. Find the second-best time.

**Hint:** Start small, create a new class with a method that first tries to print all the times, then find the best time, then the second best.

## Exercise 6:

Write a program that receives an Array of numbers and prints the array inverted and the array that contains the squares of the original numbers.

**Hint:** Consider creating two methods, one for inverting and one for creating an array of squares. Java actually has build-in support for the inverted array (as well as sorting, etc.).

## Exercise 7:

Fibonacci numbers are a sequence of integer numbers, characterized by the fact that every number after the first two is the sum of the two preceding ones:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Create a class called `FibonacciCalculator` that includes a method called `fibonacci`. Your method should take as input an integer number *n* and return the Fibonacci integer value in the n-th position of the sequence (e.g., `fibonacci(4)=3`). In order to solve the program we will use recursion, i.e., call the `fibonacci` method within the `fibonacci` method. In particular, consider that `fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)` and we know that `fibonacci(1)= fibonacci(2)=1.` From your main method run a *for* loop that prints the first 10 Fibonacci numbers. (This is a very inefficient implementation of the Fibonacci method and not recommended for *n* bigger than 45. More generally, Fibonacci numbers become quickly very large and computers cannot represent them - but the method takes tooooo long to compute anyway by then ☺).

**Hint:** Do not try to write the entire program in one go. It is much easier to write a small piece and test it, then write another small piece and test it. For example, start by writing just a skeleton of your method that returns *n* (instead of the Fibonacci at position *n*) and test that the *for* loop in your main method works well. Then add the code in your Fibonacci method to do a simple test for the value of *n* (if it is one of the basic cases 1 or 2), before calling your function for *n-1* and *n-2*. You should test your program with simple test inputs.