Software Engineering, LAB 1: Test

1. Incremental development

1) Verify your *JUnit* installation and realize the exercise incrementally, code a little test a little, using *JUnit* 4 for whatever test and after each single function's implementation.

2) Define, in a class named "Worker", a simple static method named "conversion", which receives an integer value as parameter and returns half of that value if the value is even, or three times plus one of that value if the value is odd.
Test its behavior.

   Now, write a second method, named "sequence", with an integer parameter named "startingValue", able to iteratively apply the first method ("conversion") and determine how long is the sequence of repeated calls of "conversion" until you get the value 1, starting from whatever integer value ("startingValue") > 2 (for "repeated calls" we mean that the returned value of the first "conversion" call is used as input for the next "conversion" call, and so on).
Test its behavior and test that if the starting value is <=2 the "sequence" method throws an exception.

   Example of sequence(10):

   Starting value = 10

   Obtained values through "conversion": 10 => 5 16 8 4 2 1

   Length: 6

3) Refactor your class, so that "sequence" has no parameter anymore, but the "startingValue" is defined as field of "Worker" and is initialized through the constructor:

```
Worker worker = new Worker(10);
worker.sequence();   // == 6
```

   You need a new object for each new sequence.

   Adapt your tests, to verify that this new version still works correctly.

4) Write a new class "SequenceCache", able to return the sequence for each value, using an internal Map to cache the pairs <value, Worker object>.

The value is specified as integer parameter of the method “length”.
If for a certain value the corresponding Worker already exists in the Map, the “length” method internally calls the Worker method “sequence” on it and returns the sequence value.

If, however, there is no Worker, the class creates a new corresponding worker first, adds it to the Map, and then calls the “sequence” method on it.

```
SequenceCache sc = new SequenceCache();
sc.length(10);  // == 6
```

Test the behavior of the new class “SequenceCache”, also considering exceptional cases.

2. Fraction

1) Define the class “Fraction”, representing a fraction as pair of longs. Implement the 4 base operations, starting from the addition.
   Implement the “simplify” operation, using the greatest common divisor (gcd) function.
   Remember: provide the equals method for Fraction, able to compare for equality two distinct, but equal, objects.

   Note:
   the greatest common divisor (gcd), also known as the greatest common factor (gcf), highest common factor (hcf), or greatest common measure (gcm), of two or more integers (at least one of which is not zero), is the largest positive integer that divides the numbers without a reminder. For example, the GCD of 8 and 12 is 4.

   gcd(u,0) = u
   gcd(u,v) = gcd (v,u)
   gcd (u,v) = gcd (v,u%v)

   Least common multiple:
   lcm(u,v) = u * v / gcd (u,v).

2) Implements in *JUnit* all test demonstrating the correct behavior of the class, also considering exceptional cases (like division by zero).