Java exercises

Please see the separate sheet about running Eclipse if you are not familiar with the Eclipse IDE.

Task 1

Write a Java method to compute and print the list of prime numbers less than a given parameter. The method signature should be as follows:

```
void printPrimes (int max)
```

Task 2

Write a Java method to compute and return the Nth Fibonacci number, which is a sequence of numbers where each number is the sum of the preceding two (see https://en.wikipedia.org/wiki/Fibonacci_number if you can't remember the details). The method signature should be as follows:

```
int computeFibonacci (int n)
```

Task 3

Implement a method **`computeScore`** that correctly computes and returns the score that you would receive in the game of Scrabble[1] for the string given as a parameter – that is, you must add up the score of each letter in the string and return the total value. Refer to the following table of values for each letter:

A: 1	B: 3	C: 3	D: 2	E: 1
F: 4	G: 2	H: 4	I: 1	J: 8
K: 5	L: 1	M: 3	N: 1	O: 1
P: 3	Q: 10	R: 1	S: 1	T: 1
U: 1	V: 4	W: 4	X: 8	Y: 4
Z: 10				

If you encounter any characters other than those 26 letters, then they should not affect the total score, but **you should be sure to consider both upper case and lower case letters**.

The signature for the method must be as follows:

```
int computeScore (String word)
```

[1] https://en.wikipedia.org/wiki/Scrabble or http://www.scrabble.com/

Task 4

Design and implement a class `Monster` representing a (simplified) monster in a monster-battling game. A monster includes a **type** (a String), a number of **hit points** (an int), a number of **attack points** (an int), as well as list of **weaknesses** (i.e., types against which the monster is particularly weak in battle, represented as a String[]).

Define a **Monster** class with appropriate fields with correct access modifiers, as well as getter methods for all fields and an appropriate constructor.

You should also define the following two methods which are used when monsters battle each other:

- `public boolean isWeakAgainst (String otherType)`
  - This method should return **true** if **otherType** is included in this Monster's **weaknesses** list, and **false** if it is not. Don't forget to use **.equals()** to compare String values instead of "=="
- `public void removeHitPoints (int pointsToRemove)`
  - Removes the indicated number of hit points from the current monster. If the hit point value becomes negative, then it should be set to zero.
- `public boolean attack (Monster otherMonster)`
  - This method is called when the current monster attacks the other monster. It should proceed as follows:
    - If **otherMonster** is actually this monster (use "==" to check), return false immediately
    - If either the current monster or the other monster is knocked out (i.e., hit points == 0), return false immediately
    - Otherwise, check if the other monster is weak against the current monster's type (use **isWeakAgainst()**)
      - If it is not weak, remove only the current monster's attack points from the other monster's hit points (use **removeHitPoints**)
      - If the other monster is weak, remove the attack points plus an additional 20 points (again, use **removeHitPoints**)

Task 5

Based on the **Monster** class created above, you should **refactor** it into a set of classes that are able to represent every type of monster as its own class, instead of using the **type** field to distinguish them. The information that is common to all monsters will be retained in the **Monster** class, which will be made **abstract**, while other information that is relevant only to one of the monster types will be put into the appropriate subclass. The subtypes should be **WaterMonster, FireMonster,** and **ElectricMonster**, with the following properties:

- A **FireMonster** has type "Fire" and is weak against Water
- A **WaterMonster** has type "Water" and is weak against Fire and Electric
- An **ElectricMonster** has type "Electric" and has no weaknesses

dodge()

Add an abstract **dodge()** method to the parent **Monster** class – this method should return a **boolean** value and will be implemented in the subclasses to implement the modified **attack()** behaviour described below. The method should have a **protected** access modifier.

The required behaviour for **dodge()** in each subclass is as follows – you should add any necessary fields to each subclass to implement this behaviour:

- **FireMonster:** this method should alternatively return **true** and **false** – that is, the first time it is called, it should return **true**, the next call should return **false**, and so on
- **WaterMonster:** this method should return **true** if the monster's hit points are at least 100, and **false** if they are less than 100.
- **ElectricMonster:** this method should always return **false** – that is, an electric monster should never dodge when attacked.

attack()

The final piece of refactoring is to modify the **attack()** method of **Monster** to use **dodge()** as follows:

- First, call **dodge()** on the monster being attacked.
- If the result is **false**, the attack behaviour as before is implemented.
- If the result is **true**, no hit points are removed from the monster being attacked, but 10 hit points are removed from the monster doing the attacking. The same rules apply here – if the monster's HP goes below zero, then it should be set to zero.

Task 6

Your task is to write a program to check whether a given credit card number is valid or invalid. You can read more about the process of checking credit card numbers at several websites, including http://www.validcreditcardnumber.com/; the description below is a summary of the properties that we will be using for this lab.

The first several digits of the card number indicate the issuing network of the card:

- **Visa** card numbers all start with **4**
- **American Express** card numbers all start with **34** or **37**
- **MasterCard** card numbers all start with a number between **51 and 55 (inclusive**), or with a number between **2221 and 2720 (inclusive)**

The remainder of the card number is allocated by the card issuer. In general, a valid credit card number can be anywhere between 13 and 19 digits; however, depending on the issuer, the valid lengths can vary:

- All **Visa** card numbers are of length **13, 16 or 19**
- All **American Express** card numbers are of length **15**
- All **MasterCard** card numbers are of length **16**

Finally, the final digit of all credit card numbers is a **check digit** – that is, a digit computed from the other digits of the card number. Including this digit means that simple errors in manually entering a card number (such as a single mistyped digit or permutations of successive digits) will be caught because the card will not be validated. For credit cards, the check digit is computed using the Luhn Algorithm (https://en.wikipedia.org/wiki/Luhn_algorithm).

To test whether a credit card number is valid, proceed as follows (adapted from the Wikipedia page above):

1. **Proceeding from the rightmost digit and moving left**, double the value of **every second digit**. If the result of this doubling is greater than 9, then subtract 9 from the product.
2. Take the sum of the resulting digits.
3. The credit card number is valid if and only if the sum is a multiple of 10.

As a concrete example, consider the number “79927398713”. The validation process would proceed as shown in the following table.

Account number	7	9	9	2	7	3	9	8	7	1	3
Double every other	7	18	9	4	7	6	9	16	7	2	3
Digits to sum	7	9	9	4	7	6	9	7	7	2	3

The sum of the resulting digits is 7+9+9+4+7+6+9+7+7+2+3=70, so the number is valid according to the Luhn algorithm.

What you need to do

You must implement a method to validate a credit card number. The method signature should be as follows:

```
boolean checkCardNumber (String cardNumber)
```

Your method should carry out all of the checks listed on the previous page and should return **true** if the card number is valid and **false** if it is not. You can use the sample (fictional) credit card numbers from websites such as http://www.getcreditcardnumbers.com/ to test your code. Be sure to test on both valid and invalid numbers!