|  |  |  |  |
| --- | --- | --- | --- |
| Nr | Points | Problem | Notes |
|  | 4 | Write a small application that reads a list of personalities from a csv file (Firstname, Lastname, dob, dod) and stores the information in objects. The application should filter out duplicated entries and output the list of objects to the console in a human readable manner:  "*Firstname Lastname (dob-dod)*". | The code that reads the lines from the file and the input file are provided  See provided source folder |
|  | 4 | Write a program that prints values from 1 to custom *n* number, provided as an argument to the application. Print the *PRIME* keyword near each prime number. (1, 2-*PRIME*, 3-*PRIME*, 4, …. , 100). | Run from the command line and provide the argument *n*. |
|  | 6 | The customer contacts a reseller when he wants to buy a new domain. In order to complete the customers wish the reseller contacts a registrar and triggers the create domain process. The registrar knows how to talk to the registry in order to create the customer’s domain. Also the customers must provide some details for the domain they want to create: the name of the domain, owner details and hosts. Use OO techniques to model this problem.  Provide:   * Java classes * UML Class diagram * UML Sequence diagram | Hint:  You can use Astah or MS Visio (not mandatory) for the UML diagrams. |
|  | 6 | Create a class called ConnectionManager that manages a fixed array of Connection objects. The client programmer must not be able to explicitly create Connection objects, but can only get them via a method in ConnectionManager. When the ConnectionManager runs out of objects, it returns a null reference. Test the classes in main(). |  |

General requirements:

* Follow at least the general Java naming conventions as outlined here: <http://google-styleguide.googlecode.com/svn/trunk/javaguide.html#s5-naming>
* Whether Javadoc should be present or not depends on the mentor, whether they rely on clean code or not. If Javadoc is mandatory, it should be used on all public and protected methods of a class, except getters and setters. Try to add information about your reasoning in the Javadoc when possible instead of restating what is already obvious from the method signature.
* Don’t be afraid to use longer explanatory names for your variables and methods. The only place where it’s ok to use one letter names is in for loops. In real world situations, code will be written once, but read at least a dozen time over the life time of a project, so it needs to be optimized for readability and understanding.
* Every solution should be checked in a separate folder, and the folder structure should be the following
  + week[x]/p[x]/
* Source structure should the maven default folder structure
  + - week[x]/p[x]/
  + └ src
  + └ main
  + └ java/[full\_package]
  + └ resources
  + └ test
  + └ java/[full\_package]
  + └ resources
* Java packages should follow the Java package conventions:
  + - com.iquestgroup.remotelearning.week[x].p[x].[package\_name]
* Don't check in binary files (only in special cases) like .class, .jar, .exe etc.
* Don't check in local IDE specific configuration files.
* Format your code according to the Sun's coding convention (should be the default format in your IDE - so use it)
* Java doc is mandatory only in interfaces, so use it only when needed. Try to use clean code.
* Don't ever forget the commit message.
* Class & method responsibility: Do one job (no more or less), avoid doing everything in main
* Write unit tests \[After week 5\]
* Enjoy coding.

Week1/p1/src/main/java/com/iquestgroup/remotelearning/week1/p1/Class.java