2DV600 - Assignment 2

Grading
Your submission will receive a grade from A to F where F is Failed.
You are allowed to improve your work after the initial submission before the deadline.
Please note that higher grades are subject to the following criteria:
i) correct functionality, ii) quality of the solution, iii) quality of code documentation, code structure, and organisation. You may use the attached coding conventions document as a guideline to document your code. **Don't forget to submit your assignments on time!**

The grade is final, i.e., you will not get an opportunity to correct/improve after grading.

Your answers should be your own! You are not allowed to copy code from other students, books articles, blogs, wikis or any other source! Each submission will pass through a plagiarism/clone detection system before correction. **If plagiarism is detected, the assignment will be failed and a formal investigation will be initiated.**

Help
Charilaos Skandylas is the teaching assistant to help you with problems related to the practical assignments. Don't hesitate to contact him if you need help. He will be present during laboratory lessons, see the time plan, to assist you in the assignments' tasks. Moreover, you may use the Moodle forums for help and questions related to practical assignments.

Preliminaries
1. We recommend using Eclipse but you are free to use any IDE or tool of your choice to write Java code.
2. Using eclipse, create a *package* with the name `YourLnuUserName_assign2` inside the project `2DV600` that you created for the assignment 1. For example, it might look something like `wo222ab_assign2`.
3. Save all program files from the exercises in this assignment inside the package `YourLnuUserName_assign2`.

Programming Exercises
Notice: you are not allowed to change anything (not even package names) in the classes and interfaces provided to you as part of following exercises.

Exercise 1
Each character in a role-playing game is represented by a *name*, a *class*, and a *level*. The level of each character corresponds to an integer value, while the name and class are strings.
Each character, additionally has four attributes: Strength, Agility, Intelligence and Wisdom.
Furthermore, they belong to one of the four following classes: Mage, Warrior, Rogue, Cleric.
Each class has a primary attribute which is intelligence for mages, strength for warriors, wisdom for clerics and agility for rogues.

Department of Computer Science
SE-351 95 Växjö • *Visiting address* B-House, Campus Växjö • *Telephone* +46 470 708460 • *Email* jesper.andersson@lnu.se • Lnu.se

Mages and clerics can learn spells while warriors and rogues can train in abilities.
Note that mages and clerics cannot train in abilities while rogues and warriors cannot learn spells. Mages and clerics can only learn spells that are aimed for their classes and similarly, warriors and rogues can only learn abilities targeted for their own classes.

A spell has a name and a school, mage schools include evocation and alteration, while cleric schools include restoration and divination. An ability has a name and a proficiency, warrior proficiencies include athletics and survival, while rogue proficiencies include acrobatics and stealth.

A party is composed of any number of characters of each class.

Define and implement the following classes:
An Attributes class with the following methods:
(i) getStrength, that returns the strength score
(ii) getAgility, that returns the agility score
(iii) getIntelligence, that returns the intelligence score
(iv) getWisdom, that returns the wisdom score

A Character class with the following methods:
(i) getName, returning the character's name
(ii) getClassName, returning the character's class name
(iii) getLevel, returning the character's level
(iv) getPrimaryAttribute, returning the character's primary attribute according to their class

A Spell class with the following methods:
(i) getName, which returns its name
(ii) getSchool, which returns its school
An Ability class with the following methods:
(i) getName, that returns its name
(ii) getProficiency, that returns the proficiency it is associated with.

A Mage class that extends the Character class with the following methods:
(i) learnSpell, that takes a Spell instance as an argument and adds it to the known spells if its school is allowed for the Mage class.
(ii) getKnownSpells, that returns a list of Spell instances that the mage already knows

A Cleric class that extends the Character class with the following methods:
(i) learnSpell, that takes a Spell instance as an argument and adds it to the known spells if its school is allowed for the Cleric class
(ii) getKnownSpells, that returns a list of Spell objects that the cleric already knows

A Warrior class that extends the Character class with the following methods:
(i) learnAbility, that takes an Ability instance as an argument and adds it to the known abilities if its associated proficiency is one that the Warrior class can train in.

(ii) getKnownAbilities, that returns a list of abilities that the warrior already knows

A Rogue class that extends the Character class with the following methods:
(i) learnAbility, that takes an Ability instance as an argument and adds it to the known abilities if its associated proficiency is one that the Rogue class can train in.
(ii) getKnownAbilities, that returns a list of abilities that the rogue already knows

A Battle class with the following static methods:
(i) printStatistics(Character party[]), that prints the average level of the party's characters, the total number of spells, the total number of mage and cleric spells separately,
the number of abilities and the total number of warrior and rogue abilities separately
(ii) resolve(Character party1[], Character party2[]), that returns a winning party. The rules for resolution are:

- If a party has more characters than the other it wins.
- If both parties have the same amount of characters, the parry with the highest average party level wins.
- If both the number of characters and the average party level are equal then the party whose characters have the highest total sum of primary attributes wins, otherwise the battle is a tie.

(iii) singleCombat, that takes two Character instances of the same class and follows similar rules to decide which of the two characters wins.
The rules are as follows:

- If one of the two characters is higher level than the other, then that character wins.
- If both characters are of equal level, then the one with the highest primary attribute wins.
- Should both characters have equal level and primary attributes, then the one with the most known spells in the case of a mage or cleric characters, and the one with the most known abilities in the case of warrior or rogue characters wins.
- Otherwise the single combat is a tie.

Note: the average party level is computed by summing up the level of each character in the party and dividing by the party size.

Exercise 2
Download the zipped directory **two_dv600.zip** from the assignment home page. The directory contains an abstract class AbstractIntCollection and two interfaces IntList and IntStack. The abstract class contains support for developing array-based data structures. The two interfaces define the functionality of an integer list and an integer stack. Your task is to implement the two interfaces by inheriting the support provided by the abstract class and by adding the code required for each individual data structure. That is, provide two classes ArrayIntList and ArrayIntStack with the following signatures.

```
public class ArrayIntList extends AbstractIntCollection implements IntList

public class ArrayIntStack extends AbstractIntCollection implements IntStack
```

Write also a program CollectionMain that demonstrates how the two classes can be used.

Note: The two classes must make use of the abstract class and you are not allowed to make any changes (not a single character) in either the abstract class or the two interfaces.

Exercise 3
In Exercise 2, you implemented two classes ArrayIntList.java and ArrayIntStack.java. The task is now to implement a JUnit test (CollectionTest.java) for the classes. The focus should be on verifying the correctness of each public method using maximum, minimum, just inside/outside boundaries, typical values, and error/exception values. Further, you should test only one code unit at a time, i.e., separate test methods should be defined for separate code units such as methods.

Exercise 4
First implement a queue interface and then write a JUnit test. More precisely:

1. Provide a generic implementation of the following Queue interface:

```
public interface Queue<E> extends Iterable<E> {
      int size();                          // current queue size
      boolean isEmpty();                   // true if queue is empty
      void enqueue(E element);             // add element at end of queue
      E dequeue();                         // return and remove first element.
      E first();                    // return (without removing) first element
      E last();                     // return (without removing) last element
  }
```

   Illegal operations on an empty queue (e.g., `last()`) should generate an exception.

   Notice 1: You are not allowed to use arrays or any of the existing data structures in the Java Class Library in this exercise.

2. Write a JUnit test (QueueTest.java) for the generic Queue implementation that verifies the correctness of each method in the interface. Remember to test also the "extreme cases" (operations on an empty Queue, add a huge amount of elements, etc.). Further, the testing guidelines given in exercise 3 applies to this exercise as well.

Exercise 5 – Count Words
Count words is a bigger task, divided into smaller steps for simplicity. It involves a text file **HistoryOfProgramming.txt**, which can be downloaded at the assignment home page. What we want to do is to count all words in HistoryOfProgramming.txt by adding "words" to a set.

Task 5.1
Write a program IdentifyWordsMain.java reading a file (like HistoryOfProgramming.txt) and divide the text into a sequence of words (word=sequence of letters, so you should remove digits, commas, bracket and all characters other than letters). Save the result in a new file that should be named as *words.txt*. Example:

```
Text
====
Computer programming, History of programming
From Wikipedia, the free encyclopedia (081110)
The earliest known programmable machine (that is a machine whose
behavior can be controlled by changes to a
"program") was Al-Jazari's programmable humanoid robot in 1206.

Sequence of words
=================
Computer programming History of programming
From Wikipedia the free encyclopedia
The earliest known programmable machine that is a machine whose
behavior can be controlled by changes to a
program was Al Jazaris programmable humanoid robot in
```

All exceptions related to file handling shall be handled within the program.

Task 5.2
Create a class Word, representing a word. Two words should be considered equal if they consist of the same sequence of letters and we consider upper case and lower case as equal. For example hello, Hello and HELLO are considered to be equal. The methods e*quals* and *hashCode* define the meaning of "equality". Thus, the class Word should look like the following.

```
public class Word implements Comparable<Word> {
   private String word;

   public Word(String str) { ... }
   public String toString() { return word; }
   /* Override Object methods */
   public int hashCode() { "compute a hash value for word" }
   public boolean equals(Object other) { "true if two words are equal" }
   /* Implement Comparable */
   public int compareTo(Word w) { "compares two words lexicographically"}
}
```

Note:

- If you want, you can add more methods. The methods mentioned above are the minimum requirement.
- We will use the class Word in Task 5.3 and for some tasks in the next assignment. Thus, carefully test all methods before proceeding.

Task 5.3
Create a program WordCount1Main.java doing the following:
For each word in the file word.txt

- Create an object of the class Word
- Add the object to a set of the type java.util.HashSet

- Add the object to a set of the type java.util.TreeSet

Note:

- The size of the sets should correspond to the number of unique words in the file. (Our tests gave 350 words for the file HistoryOfProgramming)
- An iteration over the words in the TreeSet should give the words in alphabetical order.