EAS Assignment Brief

Aston University
Engineering & Applied Science

CS2310 Data Structures and Algorithms with Java	Coursework 2019/20
Sylvia Wong (s.h.s.wong@aston.ac.uk)	WASS: `https://wass.aston.ac.uk/pages/viewcalendar.page.php?cal_id=410`

Assignment Brief/ Coursework Content:

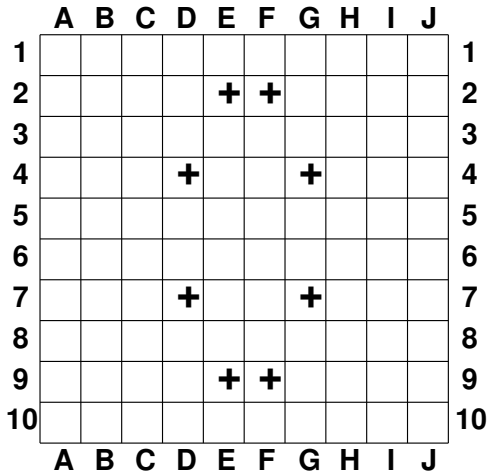
This coursework is designed to assess your achievement of the following learning outcomes of the module:

- Select and apply suitable Abstract Data Types (ADTs) within a principled software development process and ADT implementations, based on appropriate criteria.
- State Java language support for ADTs and use them for software development.

Descriptive details of Assignment:

Consider a single-player *Scrabble*-like word game where the player places letter tiles on a 10×10 game board to form valid English words whose letters can be laid out on the game board horizontally (from left to right) or vertically (from top to bottom). Each cell in the game board may hold one letter tile. Each cell is referenced by its column and row. For example, `A1` and `J10` are the cells at the top left hand corner and the bottom right hand corner, respectively.

There are eight special cells in the game board. When a letter tile is placed on a special cell, the value of the letter tile is doubled. Each special cell is marked by the symbol $+$ on the game board and they are located at `E2`, `F2`, `D4`, `G4`, `D7`, `G7`, `E9` and `F9`. The following shows the game board at the start of the game.



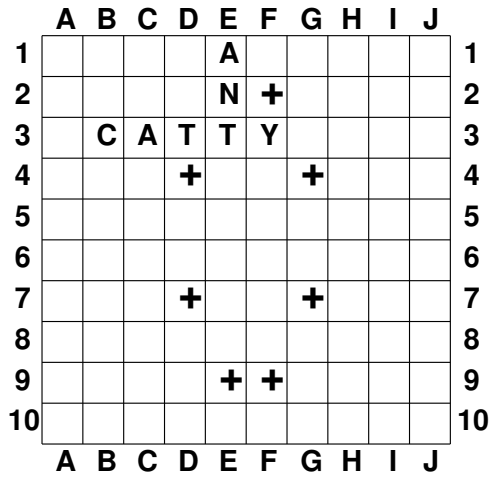
The word game has unlimited supply of letter tiles. Each letter tile contains a letter in the English alphabet and the points the letter contributes to the score of the tiles placed in forming the intended English word.

The following table shows the points assigned to each letter in the English alphabet.

Points	Letters
3	q, x, y, z
2	b, g, j, k, m, n
1	all other letters in the English alphabet

The score for each ***play*** is calculated by summing the points of the tiles placed on the game board in that ***play***. For example, if cells `B3`, `C3`, `D3` are occupied by the tiles `C`, `A` and `T`, placing the tiles `T` and `Y` in `E3` and `F3` in the next ***play*** will score 4 (i.e. $1 + 3$) points. Placing the tiles `A` and `N` in `E1` and `E2` in another ***play*** will score 5 points (i.e. $1 + 2 \times 2$).

At the start of the game, the player is given a random set of 5 tiles to make a valid English word on the game board. The player can place their tiles *anywhere* on the board to make a valid word reading either from top to bottom or from left to right. In subsequent valid '***plays***', the player must place at least one of their tiles on an unoccupied cell adjacent to a tile on the game board placed during a previous ***play***. Each valid ***play*** must lead to at least one new valid English word being introduced to the game board while ensuring that no invalid English word is introduced. A valid English word is a word that appears in the word list[1]available at `http://www-personal.umich.edu/~jlawler/wordlist`. For example, with the game board being in a state shown below, placing `O` in cell `D2` will lead to an invalid ***play*** because, even though the letter sequence `ON` is a valid English word, the letter sequence `OT` is not a valid English word.



After each ***play***, the tile rack may be refilled with a random set of tile(s).

The player may quit the game at any time, e.g. when they had enough of the game or when they cannot place any letter tile on the game board to form valid word(s) any more.

[1]A copy of the word list is available from Blackboard in the given data file named `wordlist.txt`.

Functional Requirements

F1. A player can instruct the game engine to refill the tile rack with randomly selected tiles.

The game engine will display the contents of the tile rack after the refill.

F2. A player can view the current state of the game board and the contents of the player's tile rack.

F3. A player can place some tiles horizontally or vertically on a sequence of unoccupied cells in the game board.

The game engine will display the state of the game board after the tiles have been placed on the board.

F4. A player can instruct the game engine to calculate the points for the tiles placed on the game board in a intended *play*.

The game engine will display a message stating the points to be scored.

F5. A player can instruct the game engine to check the validity of a specified *play*.

The game engine will display a message stating "Valid" or "Invalid". With a valid *play*, the game engine will also display all new English words introduced to the game board. With an invalid *play*, the game engine will also display the invalid letter sequence introduced to the game board.

Non-functional Requirements

1. The processing for Functional Requirements **F1–F4** *must* be done in linear time (i.e. $O(n)$) or less where n in each functional requirements refers to:

   (a) the size of the tile rack,
   (b) the size of the game board,
   (c) the number of tiles placed on the game board in the *play*, and
   (d) the number of tiles placed on the game board in the *play*,

   respectively.

2. The display of any system output must be clear and easy to understand.

3. The application should be robust and display appropriate messages should any run time errors (e.g. cannot place tile beyond the boundary of the game board or on an occupied cell) occur.

4. A user will interact with the system through **the given Text-based User Interface** (TUI), with all output displayed by the standard output stream.

Given Java Code

- `TUI.java`: a Java class modelling a Text-based User Interface for the intended word game application,
- `Controller.java`: a Java interface containing five abstract methods for modelling the functional requirements with each method returning a string as its output, and
- `Play.java`: a Java class modelling the information required for a play, i.e. the starting cell, the direction by which the letter tiles are to be placed on the game board, and the positions of the letter tiles on the tile rack.

The given TUI is modelled in the Java class named `TUI.java`. The constructor of class `TUI` contains a 'forever' loop which displays the following menu options:

```
Word Game

Enter the number associated with your chosen menu option.
1: Refill tile rack
2: Display game state
3: Play
4: Calculate score for an intended play
5: Check the validity of a given play
6: Exit this application
```

and waits for the user's menu option choice. Depending on the menu option chosen, the user may be prompted to enter more information. For example, menu options 3–5 will prompt the user to enter the ID of the starting cell, the direction by which the letter tiles are to be read and the positions of the letter tiles on the rack. For example, `B3, DOWN, 513` means placing the 5th, 1st and 3rd tiles on the tile rack vertically starting at cell `B3` of the game board. The TUI will reject invalid cell IDs, unrecognised directions, and unacceptable tile position sequences.

Apart from menu option 6, each menu option display the result of the operation on the console (i.e. standard output stream) in plain text format.

Further particulars

- You should use collection classes in the JCF, standard Java packages and the collection classes discussed in the lectures for your implementation whenever appropriate.

- Your programs must be compilable and executable by J2SDK 8 under command line prompt, without the use of `Eclipse` or any other IDEs.

  The API specification for the Java 2 Platform, Standard Edition, version 8 is available at `http://docs.oracle.com/javase/8/docs/api/`

- This coursework is a piece of *group work* for a team of 3 or 4 students. Inadequate participation by any team member will be penalised. Non-participation by any team member will result in a mark of zero. The module leader's decisions are final and cannot be challenged.

  Note that any team issues must be discussed with the module leader as soon as these become apparent, so that appropriate intervention or mitigation can be considered.

- Any team member may submit your team's solution to Blackboard, but **only the latest submission** from the team will be marked.

  To alleviate the potential issue of identifying a mistake in the code after a submission has been made, you will be able to submit multiple versions of the same piece of work. Note that **only the latest version** of the submission will be marked.

- In your submission, you are permitted to use **only** the methods and programming routines that:

  - are given as part of the coursework specification,
  - are from the J2SDK 8 library,
  - are part of the lecture material, or
  - are written by you.

  > N.B.: *Submissions which fails to comply with this requirement will be reported to the Associate Dean of Undergraduate Programmes as a suspected case of plagiarism.*

- Uncommented code will **not** be accepted.

  > N.B.: *Do not write too many comments neither.*
  > *It is a bad practice to write one line comment for each line of code as this makes your code more difficult to maintain and to be kept up to date.*

- You **may** be required to give a demonstration of your application.

Recommended reading / online sources:

Lawler, J. (1999), *An English Word List*. [Online] Available at: `http://www-personal.umich.edu/~jlawler/wordlist.html` [accessed 31-10-2019].

Oracle (2016), *Java™ Platform, Standard Edition 8 API Specification*. [Online] Available at: `http://docs.oracle.com/javase/8/docs/api/` [accessed 18-10-2019].

Further implementation hints

- The given data file uses the standard ASCII character set which is known as `java.nio.charset.StandardCharsets.US_ASCII` in Java.
- You might like to consider using some of the following facilities provided by J2SDK:
  - IO facilities:
    * `java.util.Scanner`
    * `java.io.BufferedReader`
    * `java.io.File`
    * `java.io.InputStreamReader`
    * `java.io.FileInputStream`
    * `java.nio.charset.StandardCharsets`
    * `java.nio.file.Files`
    * `java.nio.file.Paths`
  - facilities from the JCF in `java.util`:
    * Deque (double ended queue): `Deque`, `ArrayDeque`, `LinkedList`
    * Lists: `List`, `ArrayList`, `LinkedList`
    * Maps: `Map`, `Map.Entry`, `HashMap` and `TreeMap`
    * Sets: `Set`, `SortedSet`, `HashSet` and `TreeSet`
    * `Collection`, `Collections`
  - facility for representing a sequence of characters which supports efficient operations: `java.lang.StringBuilder`

    String concatenation is the usual way to form a new string by putting several existing strings together. However, when the resulting string becomes very long, this way of concatenating strings will become extremely inefficient.

    When dealing with long strings `java.lang.StringBuilder` would be more efficient. Method `append` in a `StringBuilder` object works well for extending strings in standalone, single-threaded Java applications.

- To facilitate your system testing, you might like to implement some tester classes or methods, e.g. using `JUnit`. This would enable special tester objects to be created for *simulating the behaviours of various test cases*, hence reducing the time required to repeatedly enter the same set of data into the system after each modification made to the system. Note that the **implementation of tester classes/methods is NOT a requirement for this coursework**.

- To tokenise a string, you might like to use the tokenisation facilities in `java.util.String` or the regular expression facilities supported by the `java.util.regex` package. For example, the following Java code:

```
String text = "Hello, world";
String[] result = text.split(",");
for (String s : result) {
    System.out.println(s);
}
```

  prints the following output:

```
Hello
world
```

  To find out more about how to use regular expressions in Java, see:

  - `java.util.regex.Pattern`: Define a regular expression (`http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html`)
  - `java.util.regex.Matcher`: Perform match operations on a character sequence by interpreting a regular expression (`http://docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html`)

- To evaluate the time efficiency of your algorithm(s), you might like to time some of the tasks performed by your application, e.g.:

  - How long does it take to initialise the application?
  - How long does it take to process a user query?

  You may use the following skeleton Java code for this purpose:

```
long startTime = (new Date()).getTime();
//TODO: the process to be timed (e.g. a method call)
long endTime = (new Date()).getTime();
long elapsedTime = endTime - startTime;
//TODO: output the recorded execution time in milliseconds
```

Key Dates:

31/10/2019	Coursework set
01/11/2019 and 06/12/2019	Supporting lab sessions regarding assessment
12/12/2019 12:00	Submission due date/time
19/12/2019 12:00	Submission cut off date/time
17/01/2020	Expected feedback return date via Blackboard

Submission Details:

Submit your work as **a single ZIP file** to Blackboard. Your ZIP file must include:

- a UML class diagram describing your detailed class design in **PDF** format,
- your Java code[2] (i.e. files with `.java` extension), and
- sample output of your application (in plain text format, i.e. files with `.txt` extension).

Submission Procedure:

1. Create a new folder named after the login name of a group member, e.g. if one of your group members has the login name `hawkins`, your folder should be named `hawkins`.
2. Copy the following to this new folder:

(a) your UML class diagram(s) in **PDF** format,

(b) sample output of your application (in plain text format, i.e. files with `.txt` extension), and

(c) your `Eclipse` project folder for this exercise which contains Java programs ONLY (i.e. files with `.java` extension). The project folder structure should be left unchanged.

Make sure that you have REMOVED ALL `.class` *files, Java doc HTML files and all hidden files and folders, e.g.* `.metadata/`, `.project`, *etc.*

3. Create a `zip` archive of this folder (e.g. using `winzip`).

   The archive **must** *be named after the login name used in Step (1).* For example, if the login name used above is `hawkins`, the name of your zip file will be `hawkins.zip`.

4. Submit, to Blackboard, the created `zip` archive.

Standard lateness penalty[a] of 10% of the awarded mark for each working day that the piece of work was submitted after the formal deadline will be applied.

[a] For more information about lateness penalty, see Section 5 of Assessment Policies (AU-RSC-17-1285-A) which is available at `https://www2.aston.ac.uk/clipp/quality/a-z/examinationsandassessmentregulations/index`.

[2] The project folder structure should be left unchanged.

Marking Rubric:

- Missing submission of detailed UML class diagram (-10%)
- Your implementation in Java. (90%)

  1. Ability to meet the stated requirements (55%), e.g.:
     - Refilling the tile rack (9%)
     - Show game board and tile rack (12%)
     - Place a sequence of letter tiles on the game board (8%)
     - Calculate points (8%)
     - Check the validity of a *play* (8%)
     - Having addressed the stated non-functional requirements adequately (10%)
  2. Program design and algorithm(s) used (35%)

     Credits will be given to:
     - appropriate class design;
     - appropriate use of collection classes in JCF, including setting up each collection object with appropriate initial capacity so as to avoid runtime resizing;
     - appropriate use of other standard Java classes;
     - appropriate use of algorithm(s) for processing and indexing the data.

- Appropriate comments (i.e. documentation, block and line comments) for your implementation and the overall layout and presentation of your programs (e.g. whether your code has been indented appropriately, whether the names of variables and classes are meaningful and conform to the standard Java conventions). (10%)