CS2310 Data Structures and Algorithms with Java
Coursework 2018/9

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.

1 Dance Show Programme Generator

Astaire Dance School is a local dance school in Birmingham which runs dance classes for 3–80 years old. Pupils are put in dance groups according to their age and ability:

1. Pre-primaries: a dance group for 3–4 years old
2. Primaries: a dance group for 5–8 years old
3. Juniors: a dance group for 9–13 years old
4. Intermediates: a dance group for 14–17 years old
5. Seniors: a dance group for 18–34 years old
6. Cabaret Girls: a dance group for 35–60 years old
7. Happy Tappers: a dance group for 60+ years old

Astaire Dance School runs a whole school dance show bi-annually. Each dance show has roughly 35–40 dances performed by a mixture of dancers from the dance groups and one or more guest performers. Performers in each dance wear a custom designed costume. Each dance has a different costume. To ensure that the performers of each dance number will have sufficient time change one costume to another, there must be $\geq N$ dances between a dancer performing in one dance and his/her next dance.

1.1 Your Task

Develop a standalone Java application which works as a single-user prototype application that acts as a dance show programme generator. This application enables a user to:

- find out the names of all performers in a specified dance;
- list all dance numbers and their performers in alphabetical order;
- given a running order of a subset of dance numbers, check whether there will be enough time for each dancer to change their costume in between their dances, and
- generate a running order of the dances for the dance show.

The intended Java application is expected to work with the following tab-separated CSV files. The first line of each CSV file shows the column titles.

- `danceShowData_danceGroups.csv`: Each data line in this file shows the name of a dance group and its members. Each dancer is known by their first name. The name of each dancer is unique.

  This file contains two columns: `Dance Group` and `Members`, separated by a tab character (i.e. `\t`). The members of each dance group are listed in no particular order and are separate by a comma.

- `danceShowData_dances.csv`: This file hosts all dances and their performers in the dance show. The dances are listed in no particular order.

  Each data line in this file shows the name of a dance and its performers. The performers may be listed by the group they belonged or by their first name.

  This file contains two columns: `Dance Name` and `Performers`, separated by a tab character (i.e. `\t`). The performers of each dance are listed in no particular order and are separate by a comma.

- `danceShowData_runningOrder.csv`: This file hosts a proposed running order for a subset of dances. The dances are to be performed in the listed order.

  Both `danceShowData_dances.csv` and `danceShowData_runningOrder.csv` have the same data format.

The above CSV files are available from Blackboard (`http://vle.aston.ac.uk/`).

Your task is to design and implement the intended dance show programme generation application using J2SDK 8.

1.1.1 Functional Requirements

1. A user can use the intended system to *list* the names of all performers in a specified dance.

2. A user can use the intended system to *list* all dance numbers and the name of the respective performers in alphabetical order.

3. Given a running order of a subset of dance numbers, a user can use the intended system to *check* whether there will be enough time for each dancer to change their costume in between their dances. The user will also be able to *view* the running order of the dances where issues lie.

4. A user can use the intended system to *generate* a running order of the dances for the dance show. In the generated running order, all performers must have sufficient time for costume change. If such a running order does not exist, the intended system may either never terminate or simply display an appropriate error message.

   Hint: *The intended system is not required to detect whether a feasible running order exists.*

1.1.2 Non-functional Requirements

1. The processing for Functional Requirements $1 - 3$ *must* be done in quadratic time (i.e. $O(n^2)$) or less where n is the number of performers to be processed in the query.
2. A user will interact with the system through the given Text-based User Interface (TUI), with all output displayed by the standard output stream.
3. The display of results for each query must be clear and easy to understand.
4. The application should be robust and display appropriate messages should any run time errors (e.g. no result found) occur.

2 Further Implementation Hints

1. The given data file uses the standard ASCII character set which is known as `java.nio.charset.StandardCharsets.US_ASCII` in Java.
2. 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.

3. To prepare the data for the intended application to use, you are likely to need to tokenise each data line after reading it from the data file. You may of course use the old-fashioned `java.util.StringTokenizer` to perform tokenisation. However, as `StringTokenizer` is "*a legacy class and its use is discouraged in new code*" (Oracle 2016), you are encouraged 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 = "this, is, a, test";
String[] result = text.split(",");
for (String s : result) {
    System.out.println(s);
}
```

prints the following output:

```
this
is
a
test
```

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`)

4. 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
```

5. 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. However, *the implementation of tester classes/methods is* NOT *a requirement for this coursework*.

3 Mark Distribution

- System design expressed as a detailed UML class diagram(s). (10%)

  > **Hint:** *In your UML class diagram(s), you will need to include detailed information, i.e. fields and methods, about each class in your Java application. Do not simply use* `Eclipse` *to generate a class diagram because some notations used in the* `Eclipse` *plugin do not conform to the UML standard adequately and hence making your design unclear.*

- Your implementation in Java. (80%)

  1. Ability to meet the stated requirements (45%), e.g.:
     - List the name of all performers in a specified dance (7%)
     - List all dance numbers and their performers in alphabetical order (8%)
     - Check the feasibility of a proposed running order (10%)
     - Generate a feasible running order of the dances (10%)
     - Having addressed the stated non-functional requirements adequately (10%)
  2. Program design and algorithm(s) used (35%)

     Credits will be given to:
     - 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 given 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%)

4 Further particulars

- This coursework is a piece of *group work* for a small team of 2 or 3 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 leaders 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 the solution to Blackboard, but **only one submission** per 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. However, **only the latest version** of the submission will be marked.

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

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

- 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.*

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

- 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.

5 References

Oracle (2016), *Java™ Platform, Standard Edition 8 API Specification*. [Online] Available at: `http://docs.oracle.com/javase/8/docs/api/` [accessed 18-10-2018].

This is an assessed exercise which contributes to 20% of the module assessment.

Deadline: 13-12-2018 (Thursday) 23:59

An electronic submission **must** be made to Blackboard **by** 23:59.

The *cut-off date* for receiving coursework submission is *20-12-2018 at 23:59*.

N.B.: *Check your submission on Blackboard to ensure that the correct set of files has been submitted. Make sure that you pressed the* SUBMIT *button to complete your submission. Simply uploading your files to Blackboard will* **not** *cause your files to be submitted.*
You are advised to take into account any potential disruption, e.g. network congestion and even network failure. Hence, you are **strongly** *advised to finish all necessary preparation for your submission in good time, rather than leaving it to the last minute.*

Submission details: 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`.